Symbolische LTL-Verifikation von Petrinetzen

Von der Fakultät für
Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Diplom-Mathematiker
Jochen Spranger

geboren am 24. 8. 1966 in Tübingen

Gutachter: Prof. Dr.-Ing. M. Heiner

Gutachter: Prof. Dr. rer. nat. K. Lautenbach

Gutachter: Prof. Dr. A. Yakovlev

Tag der mündlichen Prüfung: 7. 12. 2001

Inhaltsverzeichnis

1 Einleitung

Die heutzutage in der Praxis eingesetzten nebenläufigen Systeme werden immer größer und komplexer. Die damit einhergehende Unüberschaubarkeit der Systeme stellt ein großes Problem für die Garantierbarkeit von Verhaltenseigenschaften dar. Speziell in sicherheitskritischen Bereichen, wie z. B. im Flugzeugbau, bei Atomkraftwerken oder auch bei medizinisch-technischen Geräten, wo ein Fehlverhalten katastrophale Folgen haben kann, sind auftretende Fehler nicht akzeptabel. Daher wird gerade in solchen Bereichen eine formale Überprüfung von Systemeigenschaften angestrebt.

Die Komplexität der Systeme basiert meistens nicht allein auf deren Größe, sondern wird vor allem durch die Nebenläufigkeit der Systemkomponenten drastisch erhöht. Die Handhabung von Nebenläufigkeit und die damit verbundene Komplexitätssteigerung stellt ein zentrales Problem der Informatik dar. Im Gegensatz zu sequentiellen Systemen kann das Verhalten eines nebenläufigen Systems, selbst bei gleicher Eingabe, für jeden Ablauf variieren. Dies ist auf die jeweils mögliche unterschiedliche relative Ausführung der einzelnen Komponenten zurückzuführen.

Um erwünschte bzw. unerwünschte Eigenschaften eines Systems zu garantieren bzw. auszuschließen, eignen sich formale Methoden zur Verifikation der Systeme. Diese haben in den letzten Jahren steigenden Zuspruch erfahren, was zu respektablen Erfolgen führte [CW96]. Die Verfahren, die eine automatisierte Verifikation ermöglichen, erfreuen sich dabei verständlicherweise besonderer Beliebtheit.

Ein solches automatisierbares Verfahren ist das Modelchecking. Die bekannten Modelcheckingverfahren betrachten alle erreichbaren Zustände eines Systems und sind daher nur für Systeme mit einer endlichen Zustandsmenge geeignet. Dies ist aber keine allzu große Einschränkung, da es sich bei vielen in der Praxis interessanten Systemen um Systeme mit endlich vielen Zuständen bzw. um Systeme, die sich sinnvoll auf eine endliche Zustandsmenge abstrahieren lassen, handelt.

Der erste Schritt zu einem Modelcheckingverfahren besteht darin, das System mit einem mathematisch präzisen Formalismus zu modellieren. Die bekanntesten Formalismen sind Petrinetze [Pet62] und Prozeßalgebren, wie zum Beispiel *Communicating Sequential Processes* (kurz CSP) [Hoa85] und *Calculus of Communicating Systems* (kurz CCS) [Mil89]. Petrinetze vereinen die Vorteile einer anschaulichen graphischen Beschreibungssprache mit denen eines (analysierbaren) mathematischen Formalismus und wurden daher für die vorliegende Arbeit als Modellierungssprache gewählt.

Der größte Nachteil eines Modelcheckingverfahrens ist der mit der Größe der Zustandsmenge einhergehende Aufwand des Verfahrens. Gerade bei nebenläufigen Systemen kann die Anzahl der erreichbaren Zustände sehr groß werden, da sie exponentiell mit der Anzahl der nebenläufigen Komponenten des Systems wachsen kann. Dieser Effekt wird Zustandsraumexplosion genannt.
In den letzten Jahren haben sich symbolische Verfahren im Umgang mit großen Mengen als sehr erfolgreich erwiesen. Häufig angewandt werden die reduzierten geordneten binären Entscheidungsgraphen (kurz ROBDDs) [Bry86, Bry92]. Sie codieren Zustandsmengen als Boolesche Formeln und bieten dafür eine kompakte Darstellung und effiziente Manipulation. Daher wurden sie in der vorliegenden Arbeit als Verfahren zur Zustandsrepräsentierung gewählt. Im Gegensatz zur klassischen Zustandserzeugung, die auf der Betrachtung von Einzelzuständen basiert, erfolgt die Zustandserzeugung mit ROBDD-Operationen auf einer gleichzeitigen Betrachtung aller Zustände einer gegebenen Zustandsmenge. Dies entspricht daher einer Breitentraversierung des Zustandsraumes.
Neben einer formalen Modellierungssprache für nebenläufige Systeme und einer effizienten Möglichkeit der Zustandserzeugung benötigt man noch eine formale Spezifikationssprache zur Beschreibung der zu überprüfenden Systemeigenschaften. Dafür hat sich im Bereich des Modelcheckings die temporale Logik etabliert. Eine temporale Logik ist eine um temporale Operatoren, zur Spezifikation von zeitlichen Abhängigkeiten der Zustände, erweiterte Aussagenlogik. Die beiden Hauptklassen von temporalen Logiken bilden die linearen bzw. verzweigten temporalen Logiken [Eme90]. Bei einer linearen temporalen Logik existiert zu jedem Zeitpunkt eine eindeutige Zukunft. Linear temporal logische Formeln werden daher über linearen Abfolgen betrachtet und beschreiben damit das Verhalten der einzelnen von einander unabhängigen Zustandsabfolgen eines Systems. Bei verzweigten temporalen Logiken besitzt jeder Zeitpunkt im allgemeinen mehrere mögliche zukünftige Zeitpunkte, so daß ihre Formeln über baumartigen Strukturen interpretiert werden. Sie beschreiben damit das Verhalten von möglichen Abfolgen eines nichtdeterministischen Systems. Klassische Vertreter der linearen bzw. verzweigten temporalen Logiken sind die *Linear-time Temporal Logic* (kurz LTL) [Pnu80] und die *Computational Tree Logic* (kurz CTL) [CE81].
CTL-Modelchecker sind sehr weit verbreitet, da sich ihre Komplexität linear zur Formellänge und zur Größe des Zustandsraumes verhält. Nachteilig aber ist, daß während des Modelcheckingprozesses der komplette Zustandsraum im Speicher gehalten werden muß. Dies wird aber dadurch entschärft, daß sich durch die Fixpunktcharakterisierung einer CTL-Formel das Modelcheckingverfahren direkt auf ROBDDs und deren Operationen übertragen läßt.
LTL ist eine, vor allem bei Ingenieuren, sehr beliebte Spezifikationssprache, da sie der Denkweise entspricht, daß das Verhalten eines Systems sich aus der Menge aller möglichen Systemabläufe ergibt. Die Komplexität von LTL-Modelcheckingverfahren ist aber im Gegensatz zu CTL-Verfahren exponentiell zur Formellänge. Der sich dar-

aus ergebende Nachteil wird aber meist dadurch aufgehoben, daß es für die LTL-Modelcheckingverfahren gibt, die nur den zur Überprüfung einer LTL-Formel notwendigen Teil des Zustandsraumes erzeugen und damit im Speicher halten müssen. Solche Verfahren werden on-the-fly Verfahren genannt. Die effizienten on-the-fly Verfahren basieren alle auf einem Tiefendurchlauf des Zustandsraumes und einer damit verbundenen Einzelzustandsbetrachtung.
Diese Verfahren bilden also einen der symbolischen Behandlung von Zuständen entgegengesetzten Ansatz. Die bekannten, auf binären Entscheidungsgraphen basierenden LTL-Modelchecker [CGH94, KPR95] gehören daher auch in eine höhere Komplexitätsklasse und arbeiten nicht on-the-fly, erzeugen also zur Überprüfung einer LTL-Formel immer den kompletten Zustandsraum.
Ziel der vorliegenden Arbeit ist es, die drei erwähnten Bereiche: Petrinetze als Modellierungssprache, binäre Entscheidungsgraphen zur Repräsentation großer Zustandsmengen und linear-time temporale Logik zur Spezifikation von Systemeigenschaften zu einem effizienten symbolischen Modelcheckingverfahren zu vereinen. Dabei werden die Vorteile der einzelnen Gebiete im entstehenden Verfahren beibehalten und die sich zwischen den Gebieten ergebenden Synergien ausgenutzt. Als Endergebnis wird ein symbolisches on-the-fly LTL-Modelcheckingverfahren präsentiert, welches in der selben Aufwandsklasse arbeitet wie die klassischen Verfahren.
Da sich die drei betrachteten Bereiche unabhängig voneinander entwickelt haben, wurde in dieser Arbeit besonderer Wert darauf gelegt, die Grundlagen verständlich und weitestgehend vollständig darzustellen und die Begriffswelt so zu entwickeln, daß eine Verbindung der Gebiete einfach und einsichtig möglich ist. Dies spiegelt sich in der Arbeit durch ausführliche, die drei Bereiche behandelnde, Grundlagenkapitel wider. Die Arbeit ist wie folgt aufgebaut:

Kapitel 2: In diesem Kapitel führen wir die gängigen Begriffe der Petrinetztheorie ein. Wir beschränken uns hierbei auf eine einfache, aber in vielen Werkzeugen unterstützte Teilklasse der Petrinetze, die sogenannten sicheren Stellen/Transitions-Netze. Für diese definieren wir eine spezielle Art von Stelleninvarianten, die 1-Stellen-Invarianten. 1-Stellen-Invarianten repräsentieren Stellenmengen, bei denen in jedem erreichbaren Zustand des Petrinetzes genau eine Stelle markiert ist. Diese Eigenschaft der 1-Stellen-Invarianten nutzen wir später für eine effiziente Codierung von Zuständen mit binären Entscheidungsgraphen.

Kapitel 3: Hier legen wir die Grundlagen für die bekannteste Klasse der binären Entscheidungsgraphen, die reduzierten geordneten binären Entscheidungsgraphen (kurz ROBDDs). Neben einer theoretischen Betrachtung stellen wir hier auch eine effiziente algorithmische Behandlungen dieser Datenstruktur vor.

Kapitel 4: In diesem Kapitel werden die Petrinetze und die binären Entscheidungsgraphen zusammengeführt. Wir betrachten dabei die Codierung von Petrinetz-

zuständen mit binären Entscheidungsgraphen und die Abbildung der Petrinetzschaltsemantik auf entsprechende Operationen. Im zweiten Teil dieses Kapitels gehen wir dann auf mögliche Optimierungen bzgl. der Darstellung und der Operationen ein. Basierend auf strukturellen Informationen der betrachteten Petrinetze, hier im speziellen den sich ergebenden 1-S-Invarianten, werden Verfahren zur Bestimmung guter Variablenordnungen und kompakter Codierungen von Zuständen entwickelt.

Kapitel 5: Hier führen wir in die lineare temporale Logik ein, wobei wir besonders die automatentheoretische Sichtweise beleuchten. Wir beschreiben ein in vielen Werkzeugen verwandtes Verfahren zur Konstruktion eines Büchiautomaten aus einer LTL-Formel. Da die Größe des entstehenden Büchiautomaten exponentiell mit der Länge der LTL-Formel wächst, werden zwei Verfahren zur Reduktion der Büchiautomaten entwickelt. Des weiteren werden die klassischen Verfahren zur Verifikation von LTL-Spezifikationen betrachtet.

Kapitel 6: Dieses Kapitel bildet den Kernpunkt der Arbeit. Unter Verwendung des Büchinetzformalismus werden die Systembeschreibung und die Spezifikation der Systemeigenschaften zu einem einheitlichen Formalismus zusammengeführt. Der Erreichbarkeitsgraph der entstehenden Büchinetze entspricht dabei dem Produktbüchiautomaten. Zur Handhabung des Erreichbarkeitsgraphen der Büchinetze verwenden wir die schon bekannten binären Entscheidungsgraphen und die für Petrinetze entwickelten Optimierungen. Neben der Entwicklung eines auf zwei ineinander geschachtelten Fixpunktiterationen und der Berechnung der Vorgängerzustände einer Zustandsmenge beruhenden Verfahrens zum symbolischen LTL-Modelchecking, wird hier auch ein symbolisches on-the-fly Verfahren entwickelt, welches die Aufwandsklasse der klassischen auf Einzelzustandsbetrachtung beruhenden Verfahren erreicht.

Kapitel 7: Abschließend werden die erreichten Ziele der Arbeit zusammengefaßt und einige Ausblicke auf zukünftige Erweiterungsmöglichkeiten gegeben.

Anhang A: Hier werden die entwickelten Verfahren anhand zweier aus der Praxis kommender Beispiele mit industriell eingesetzten Werkzeugen zum LTL-Modelchecking verglichen.

Anhang B: Im zweiten Anhang stellen wir kurz den begleitend zur Arbeit entstandenen LTL-Modelchecker vor.

2 Petrinetze

Petrinetze wurden 1962 von Carl Adam Petri [Pet62] als ein Formalismus zur Beschreibung von nebenläufigen Systemen eingeführt, analog zu endlichen Automaten für sequentielle Systeme. Sie vereinen die Vorteile einer anschaulichen graphischen Beschreibungssprache mit denen eines (analysierbaren) mathematischen Formalismus.

Das wichtigste Merkmal nebenläufiger Systeme im Vergleich zu sequentiellen Systemen ist die mögliche Unabhängigkeit von Ereignissen. Zwei Ereignisse werden unabhängig genannt, falls sie sich nicht gegenseitig beeinflussen. Sie sind daher nebenläufig bzw. parallel ausführbar.

In ihrer Grundform bestehen Petrinetze aus zwei Komponenten: einem Netz und einer Initialmarkierung. Ein Netz ist ein gerichteter Graph mit zwei Arten von Knoten, so daß keine Kante zwischen zwei Knoten der selben Art existiert. Man nennt die beiden Arten von Knoten „Stellen“ und „Transitionen“. Graphisch werden Stellen als Kreise und Transitionen als Quadrate dargestellt. Stellen können Marken (inhaltslose Nachrichten) enthalten, diese werden als schwarze Punkte dargestellt. Eine Verteilung von Marken auf den Stellen nennt man Markierung, welche auch als Zustand eines Petrinetzes bezeichnet wird. Eine Transition ist bei einer Markierung aktiviert, wenn auf allen ihren Eingangsstellen mindestens eine Marke liegt. Eine aktivierte Transition kann schalten und verschiebt dabei Marken. Wenn eine Transition schaltet, entfernt sie jeweils eine Marke von allen Eingangsstellen und legt jeweils eine Marke auf jede Ausgangsstelle. Das Schalten von Transitionen spiegelt die Dynamik eines Systems wieder. Es entstehen neue Markierungen d. h. neue Zustände eines Systems. Der Startzustand eines Systems wird durch die Initialmarkierung repräsentiert. Das Verhalten eines Systems wird bestimmt durch die Menge aller von der Initialmarkierung erreichbaren Markierungen.

Von dieser Grundform des Petrinetzes haben sich im Laufe der Zeit diverse Erweiterungen ausgebildet, z. B. Netze mit Kantengewichten, Prädikat/Transitions-Netze, Petrinetze mit Inhibitorkanten und zeitbewertete Petrinetze. Nähere Einblicke zum Thema Petrinetze findet man in den Standardwerken [Rei86], [Sta90] und [DE95].

2.1 Definition von Petrinetzen

Definition 1 (Netz)
Ein *Netz* ist ein Tripel $N = [S, T, F]$ bestehend aus

1. zwei endlichen disjunkten Mengen S und T, deren Elemente *Stellen* bzw. *Transitionen* heißen und
2. einer *Flußrelation* $F \subseteq (S \times T) \cup (T \times S)$, die $dom(F) \cup cod(F) = S \cup T$ erfüllt.

Als *Netzknoten* von N bezeichnen wir alle Stellen und Transitionen des Netzes. Die Bezeichnung $dom(F)$ (bzw. $cod(F)$) beschreibt die Menge aller Netzknoten, die in einem geordneten Paar aus F an erster (bzw. zweiter) Stelle vorkommen. Ein Netz ist also im mathematischen Sinne ein bipartiter Graph ohne isolierte Knoten. Einige in der Graphentheorie bekannte Begriffe lassen sich daher übertragen.

Definition 2 (Vor- bzw. Nachbereiche)
Für einen Knoten x eines Netzes N bezeichnet

$$\bullet x := \{\, y \in S \cup T \mid (y, x) \in F \,\}$$

den *Vorbereich* von x und

$$x\bullet := \{\, y \in S \cup T \mid (x, y) \in F \,\}$$

den *Nachbereich* von x.
Für eine Menge von Netzknoten $X \subseteq S \cup T$ definieren wir

$$\bullet X := \bigcup_{x \in X} \bullet x \quad \text{und} \quad X\bullet := \bigcup_{x \in X} x\bullet\,.$$

Die Stellen eines Netzes können mit *Marken* (inhaltslosen Informationen) belegt sein. Diese Belegung definieren wir wie folgt:

Definition 3 (Markierung)
Es sei $N = [S, T, F]$ ein Netz. Jede Abbildung $m\colon S \longrightarrow \mathbb{N}$ von der Menge der Stellen S in die natürlichen Zahlen wird als *Markierung* von S bezeichnet, wobei $m(s)$ die Anzahl der Marken auf der Stelle s bezeichnet.

Jede Markierung beschreibt einen denkbaren Systemzustand (d. h. Belegung der Stellen mit Marken). Eine Stelle s heißt *markiert* durch eine Markierung m, wenn $m(s) > 0$ gilt. Man kann durch eine beliebige, aber feste Indizierung der endlichen Stellenmenge $S = \{s_1, s_2, \ldots, s_n\}$ eine Abbildung m von S nach $\mathbb{N}$ als Vektor $(m(s_1), m(s_2), \ldots, m(s_n))$ darstellen, d. h. eine Markierung m läßt sich auch als ein Vektor $m \in \mathbb{N}^{|S|}$ betrachten.

Im weiteren Verlauf dieser Arbeit wollen wir bei der Betrachtung von Markierungen nicht zwischen der Abbildungs- und der Vektordarstellung unterscheiden. Des weiteren betrachten wir Vektoren, sofern nicht anders angegeben, immer als Spaltenvektoren. Wir bezeichnen mit x^T bzw. C^T die Transponierte des Vektors x bzw. der Matrix C.

Definition 4 (Petrinetz)
Ein Tupel $\mathcal{N} = [S, T, F, m_0]$ wird *Petrinetz* genannt, wenn

1. $[S, T, F]$ ein Netz ist und
2. m_0 eine Markierung von S ist.

So definierte Petrinetze werden auch als *Stellen/Transitions-Netze* (oder kurz *S/T-Netze*) bezeichnet. Die besonders ausgezeichnete Markierung m_0 wird *Initialmarkierung* (bzw. *Anfangsmarkierung*) genannt.

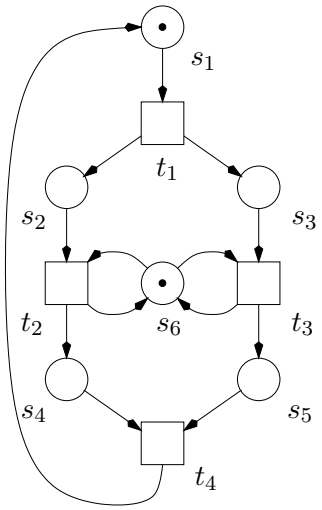


Abbildung 2.1: Stellen/Transitions-Netz

Das Petrinetz in Abbildung 2.1 besitzt

- die Stellen $\{ s_1, s_2, s_3, s_4, s_5, s_6 \}$,
- die Transitionen $\{ t_1, t_2, t_3, t_4 \}$,
- die Flußrelation $\{ (s_1, t_1), (t_4, s_1), (s_2, t_2), (t_1, s_2), (s_3, t_3), (t_1, s_3), (s_4, t_4), (t_2, s_4), (s_5, t_4), (t_3, s_5), (s_6, t_2), (s_6, t_3), (t_2, s_6), (t_3, s_6) \}$ und
- die Anfangsmarkierung $(1, 0, 0, 0, 0, 1)$.

2.2 Dynamik von Petrinetzen

Bis jetzt haben wir nur die Struktur von Petrinetzen definiert, aber noch nichts über ihre Dynamik ausgesagt.

Definition 5 (Schaltvektoren)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Zu jeder Transition $t \in T$ definieren wir die drei Abbildungen t^-, t^+ und Δt für alle Stellen $s \in S$ wie folgt:

$$t^-(s) := \begin{cases} 1, & \text{falls } s \in {}^\bullet t \\ 0, & \text{sonst} \end{cases}$$

$$t^+(s) := \begin{cases} 1, & \text{falls } s \in t^\bullet \\ 0, & \text{sonst} \end{cases}$$

$$\Delta t(s) := t^+(s) - t^-(s).$$

Analog zu den Markierungen können wir t^-, t^+ und Δt auch als Vektoren betrachten. Wie aus der Definition von Δt ersichtlich ist, sind die den Transitionen zugrundeliegenden Vektoren Δt (auch *Schaltvektoren* genannt) Elemente aus der Menge $\{-1, 0, 1\}^{|S|}$.

Definition 6 (Konzession, Schaltfähigkeit)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz und m eine Markierung von $\mathcal{N}$. Eine Transition $t \in T$ hat *Konzession* (ist *schaltfähig*) bei m, falls $m \geq t^-$ gilt.

Der Vektorenvergleich findet dabei komponentenweise statt. Für Markierungen m, m' gilt $m \leq m' :\Longleftrightarrow \forall s \in S\colon m(s) \leq m'(s)$ und $m < m' :\Longleftrightarrow (m \leq m' \wedge \exists s \in S\colon m(s) < m'(s))$. Um zu entscheiden, ob eine Transition Konzession hat, genügt es also zu überprüfen, ob alle Stellen im Vorbereich der Transition markiert sind.

Definition 7 (Schalten)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz, m eine Markierung von $\mathcal{N}$ und $t \in T$ eine Transition, die bei m Konzession hat. Dann kann t *schalten* und durch das Schalten entsteht eine neue Markierung m' mit $m' := m + \Delta t$.

Der ganzzahlige Vektor Δt beschreibt also offenbar die Markierungsänderung einer jeden Stelle, die durch das Schalten einer Transition t auftritt. Dabei werden aus dem Vorbereich der Transition Marken entfernt und dem Nachbereich Marken zugefügt. Das Schalten einer Transition ist also ein lokales Ereignis, welches nur den Vor- und Nachbereich einer Transition beeinflußt und die anderen Stellen eines Petrinetzes unverändert läßt. Die Markenzahl jeder Stelle wird dabei höchstens um eins verändert.
Die Veränderung der Markenzahl auf einer Stelle, die durch das Schalten einer Transition stattfindet, ist unabhängig von der aktuellen Markierung. Es genügt daher zur

Verfolgung der Markenverteilung auf den Stellen, die relative Veränderung bzgl. jeder Stelle und jeder Transition zu betrachten. Dies läßt sich in der sogenannten Inzidenzmatrix darstellen.

Definition 8 (Inzidenzmatrix)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Die *Inzidenzmatrix* $C: S \times T \longrightarrow \{-1, 0, 1\}$ von $\mathcal{N}$ ist wie folgt definiert:

$$C(s,t) := \begin{cases} 1, & \text{falls } s \in t\bullet \setminus \bullet t \\ -1, & \text{falls } s \in \bullet t \setminus t\bullet \\ 0, & \text{sonst.} \end{cases}$$

Analog zu der Vektordarstellung von Markierungen läßt sich die Inzidenzmatrix, wie der Name schon sagt, als Matrix darstellen. Die Matrixdarstellung hängt dabei natürlich von einer festen Indizierung der Stellen und Transitionen ab. $C(s_i, t_j)$ repräsentiert dabei den Eintrag in der i-ten Zeile und j-ten Spalte der Matrix.
Die Inzidenzmatrix des Petrinetzes in Abbildung 2.1 ist:

$$\begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{pmatrix} \text{-1} & 0 & 0 & 1 \\ 1 & \text{-1} & 0 & 0 \\ 1 & 0 & \text{-1} & 0 \\ 0 & 1 & 0 & \text{-1} \\ 0 & 0 & 1 & \text{-1} \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Der Eintrag $C(s,t)$ definiert die Veränderung der Markierung auf der Stelle s, welche durch das Schalten der Transition t stattfindet. Unter Verwendung der Schreibweise aus Definition 5 läßt sich C auch als die Matrix $(\Delta t_1, \Delta t_2, \ldots, \Delta t_k)$ mit $k = |T|$ definieren.
Jede Transition t definiert eine Relation $\xrightarrow{t}$ auf Markierungen. Dabei drückt $m \xrightarrow{t} m'$ die Tatsache aus, daß t bei einer Markierung m Konzession hat und daß m durch das Schalten von t in die Markierung m' überführt wird. Für diese Relationen gilt, daß m' durch die Angabe von m und t eindeutig bestimmt ist. Die Relationen sind also rechtseindeutig.

Definition 9 (Erreichbarkeitsrelation)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Die Menge aller endlichen Folgen von Elementen aus T bezeichnen wir mit T^*. Darunter befindet sich auch die leere Folge ϵ. Für zwei Markierungen m, m' von $\mathcal{N}$ und einer Folge $\delta \in T^*$ definieren wir die Relation $m \xrightarrow{\delta} m'$ induktiv durch:

1. $m \xrightarrow{\epsilon} m' \;:\Longleftrightarrow\; m = m'$

2. $m \xrightarrow{\delta t} m' \;:\Longleftrightarrow\; \exists m'' \colon (m \xrightarrow{\delta} m'' \wedge m'' \xrightarrow{t} m')$.

Des weiteren definieren wir die *Erreichbarkeitsrelation* $\xrightarrow{*}$ von $\mathcal{N}$ durch:

$$m \xrightarrow{*} m' \;:\Longleftrightarrow\; \exists \delta \in T^* \colon m \xrightarrow{\delta} m'.$$

Wenn $m \xrightarrow{*} m'$ im Petrinetz $\mathcal{N}$ gilt, sagen wir, m' ist *erreichbar* von m in $\mathcal{N}$.

Wir können den Begriff der Schaltfähigkeit auf Transitionsfolgen erweitern. Eine Transitionsfolge δ ist unter einer Markierung m schaltfähig (bzw. hat Konzession), falls eine Markierung m' existiert mit $m \xrightarrow{\delta} m'$.

Definition 10 (Markierungsmenge, Zustandsmenge)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Wir bezeichnen mit $\mathcal{R}_\mathcal{N}(m)$ die Menge aller von einer Markierung m in $\mathcal{N}$ erreichbaren Markierungen.

$$\mathcal{R}_\mathcal{N}(m) := \{\, m' \mid m \xrightarrow{*} m' \,\}$$

Wir wollen im weiteren die von der Anfangsmarkierung m_0 aus erreichbaren Markierungen $\mathcal{R}_\mathcal{N}(m_0)$ als *Zustände* des Petrinetzes $\mathcal{N}$ bezeichnen.

2.3 Beschränktheit von Petrinetzen

Von zentraler Bedeutung für die Petrinetze ist die Frage der Endlichkeit bzw. Unendlichkeit der Zustandsmenge. Die Handhabung von Systemen mit unendlichen Zustandsmengen erweist sich meist als sehr kompliziert. Dies drückt sich z. B. in einer sehr beschränkten Anzahl meist aussageschwacher Analyseverfahren aus.

Definition 11 (Beschränktheit)
Ein Petrinetz $\mathcal{N} = [S, T, F, m_0]$ heißt *beschränkt*, wenn die Menge der von der Anfangsmarkierung m_0 aus erreichbaren Markierungen $\mathcal{R}_\mathcal{N}(m_0)$ endlich ist. Ansonsten heißt es *unbeschränkt* .

Mit anderen Worten ist ein Petrinetz $\mathcal{N}$ genau dann beschränkt, wenn nur endlich viele Zustände vom Ausgangszustand m_0 durch Schalten von Transitionen erreichbar sind.
Es stellt sich nun die Frage nach einem Verfahren, welches für ein gegebenes Petrinetz entscheidet, ob das Petrinetz beschränkt ist und damit eine endliche Anzahl erreichbarer Zustände hat.
Die sich aus der Definition des Beschränktheitsbegriffes direkt ergebende Idee, einfach alle Zustände durch das Schalten der Transitionen zu erzeugen, führt zu keiner praktikablen Lösung, da dieser Ansatz für unbeschränkte Petrinetze nicht terminiert.

Wir suchen also ein zusätzliches Kriterium, welches uns erlaubt, die Erzeugung aller erreichbaren Zustände abzubrechen, sobald ersichtlich ist, daß das Petrinetz unbeschränkt ist.
Um ein solches Kriterium zu erhalten, betrachten wir im folgenden den Zusammenhang zwischen der Anzahl der Zustände und der Anzahl der Marken auf den einzelnen Stellen.

Satz 1 (Markenzahl und Beschränktheit)
Ein Petrinetz $\mathcal{N} = [S, T, F, m_0]$ ist genau dann beschränkt, wenn es für jede Stelle s eine Zahl $n_s \in \mathbb{N}$ gibt, so daß für alle $m \in \mathcal{R}_\mathcal{N}(m_0)$ gilt: $m(s) \leq n_s$.

Beweis: Ist $\mathcal{N}$ beschränkt, so existieren nur endlich viele erreichbare Markierungen. Für jede Stelle $s \in S$ gibt es daher ein $n_s := \max\{\, m(s) \mid m_0 \xrightarrow{*} m \,\}$.
Existiert andererseits für jede Stelle s eine obere Schranke n_s für die Anzahl von möglichen Marken auf s, so kann es nur maximal $\prod_{s \in S}(n_s + 1)$ verschiedene Markierungen geben. □

In einem beschränkten Petrinetz existiert also eine obere Schranke für die Anzahl der Marken auf jeder Stelle. Der Zusammenhang zwischen der Beschränkung der Markenanzahl auf den Stellen und der Endlichkeit der erreichbaren Markierungen ermöglicht uns einen neuen Zugang zum Beschränktheitsbegriff.
Um dies in einem praktikablen Beschränktheitstest auszunutzen, müssen wir uns zuerst noch näher mit den Vorgängen beim Schalten von Transitionen bzw. Transitionsfolgen beschäftigen.
Schaltet eine Transition t bei einer Markierung m, so gilt:

$$m' = m + \Delta t.$$

Daher folgt beispielsweise für die Transitionsfolge $\delta = t_1 t_3 t_1 t_2 t_1 t_3$ mit $m \xrightarrow{\delta} m'$:

$$m' = m + \Delta t_1 + \Delta t_3 + \Delta t_1 + \Delta t_2 + \Delta t_1 + \Delta t_3 = m + 3 \cdot \Delta t_1 + 1 \cdot \Delta t_2 + 2 \cdot \Delta t_3.$$

Wir definieren $\Delta\delta$ für eine Transitionsfolge $\delta = t_1 t_2 \ldots t_n$ als $\Delta\delta := \sum_{i=1}^{n} \Delta t_i$.
Faßt man nun die Häufigkeit des Vorkommens der einzelnen Transitionen in einer Transitionsfolge δ in einem Vektor $\Psi(\delta)$ zusammen, erhält man den sogenannten Parikhvektor einer Transitionsfolge.

Definition 12 (Parikhvektor)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz und $\delta \in T^*$ eine endliche Transitionsfolge. Der *Parikhvektor* $\Psi(\delta)\colon T \longrightarrow \mathbb{N}$ von δ ordnet jeder Transition $t \in T$ die Anzahl ihres Vorkommens in δ zu.

Kombiniert man den Parikhvektor einer Transitionsfolge mit der Inzidenzmatrix, so erhält man folgende wichtige Aussage:

Satz 2 (Markierungsgleichung)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz und m, m' zwei Markierungen von $\mathcal{N}$. Für jede endliche Transitionsfolge δ mit $m \xrightarrow{\delta} m'$ gilt folgende Gleichung:

$$m' = m + C \cdot \Psi(\delta).$$

Diese wird als Markierungsgleichung des Petrinetzes $\mathcal{N}$ bezeichnet.

Beweis: Induktion über die Länge von δ.

IA ($\delta = \epsilon$) : Hier gilt $m = m'$ und $\Psi(\delta) = \vec{0}$.

IS ($\delta = \tau t$): Sei $t \in T$ und $\tau \in T^*$ mit $m \xrightarrow{\tau} m'' \xrightarrow{t} m'$. Dann gilt:

$$\begin{array}{rcl} m' & = & m'' + \Delta t \\ & = & m'' + C \cdot \Psi(t) \\ & \overset{(\mathsf{IH})}{=} & (m + C \cdot \Psi(\tau)) + C \cdot \Psi(t) \\ & = & m + C \cdot (\Psi(\tau) + \Psi(t)) \\ & = & m + C \cdot \Psi(\tau t) \\ & = & m + C \cdot \Psi(\delta) \end{array}$$

□

Die Markierungsgleichung läßt sich auch unter Verwendung von $\Delta\delta$ als $m' = m + \Delta\delta$ schreiben.
Aus der Markierungsgleichung erhält man ein notwendiges, aber nicht hinreichendes Kriterium für die Erreichbarkeit von Markierungen. Dies ist im folgenden Korollar ausgedrückt.

Korollar 1 (Notwendiges Kriterium für die Erreichbarkeit von Markierungen)
Wenn eine Markierung m eines Petrinetzes $\mathcal{N}$ von der Anfangsmarkierung m_0 erreichbar ist, dann hat die Gleichung

$$C \cdot x = m - m_0$$

eine Lösung für x über $\mathbb{N}$.

Beweis: Ist die Markierung m von m_0 erreichbar, so existiert eine Transitionsfolge δ mit $m_0 \xrightarrow{\delta} m$. Der Parikhvektor $\Psi(\delta)$ der Transitionsfolge δ ist eine positive ganzzahlige Lösung des Gleichungssystems. □

Mit anderen Worten besagt das Korollar, daß, falls die Gleichung für eine Markierung m keine Lösung über $\mathbb{N}$ besitzt, m definitiv nicht erreichbar ist.
Eine weitere Konsequenz der Markierungsgleichung ist, daß die von einer schaltfähigen Transitionsfolge erreichbare Markierung nur von der Häufigkeit des Auftretens der

einzelnen Transitionen in der Transitionsfolge abhängt. Jede schaltfähige Permutation einer Transitionsfolge führt zur selben Markierung.
Als nächstes zeigen wir: Falls eine Transitionsfolge bei einer Markierung schaltfähig ist, so ist sie es auch unter jeder größeren Markierung.

Satz 3 (Monotonie des Schaltens)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz und m, m' zwei Markierungen von $\mathcal{N}$. Aus $m \xrightarrow{\delta} m'$ folgt $(m + l) \xrightarrow{\delta} (m' + l)$ für alle $l \in \mathbb{N}^{|S|}$.

Beweis: Induktion über die Länge von δ.

IA ($\delta = \epsilon$) : Die leere Transitionsfolge ist bei jeder Markierung schaltfähig.

IS ($\delta = \tau t$): Sei $t \in T$ und $\tau \in T^*$ mit $m \xrightarrow{\tau} m'' \xrightarrow{t} m'$. Unter Verwendung der Induktionshypothese erhalten wir $(m + l) \xrightarrow{\tau} (m'' + l)$. Da t bei m'' schaltfähig ist, folgt, daß dies auch auf $m'' + l$ zutrifft. Durch die weitere Anwendung der Definition des Schaltens und $m' = m'' + \Delta t$ erhalten wir $(m'' + l) \xrightarrow{t} (m'' + l + \Delta t)$. Daraus folgt $(m'' + l) \xrightarrow{t} (m' + l)$ und damit $(m + l) \xrightarrow{\tau t} (m' + l)$. □

Die Markierungsgleichung zusammen mit der Monotonie des Schaltens gibt uns ein Kriterium zur Entscheidung der Unbeschränktheit von Petrinetzen in die Hand.

Satz 4 (Beschränktheitskriterium)
Ein Petrinetz $\mathcal{N} = [S, T, F, m_0]$ ist genau dann unbeschränkt, wenn eine Transitionsfolge $\delta \in T^*$ und zwei Markierungen $m^0, m^1 \in \mathcal{R}_\mathcal{N}(m_0)$ mit $m^0 \xrightarrow{\delta} m^1$ und $m^1 > m^0$ existieren.

Beweis: „⇐“: Aus $m^0 \xrightarrow{\delta} m^1$ mit $m^1 > m^0$ folgt $\Delta\delta(s) \geq 0,\ \forall s \in S$. Das Lemma der Monotonie des Schaltens liefert uns

$$m^0 \xrightarrow{\delta} m^1 \xrightarrow{\delta} m^2 \xrightarrow{\delta} m^3 \xrightarrow{\delta} \cdots$$

und damit $m^i = m^0 + i \cdot \Delta\delta$. Des weiteren folgt aus $m^1 > m^0$, daß mindestens eine Stelle s mit $\Delta\delta(s) > 0$ existiert. Es ergibt sich daher aus $m^i(s) = m^0(s) + i \cdot \Delta\delta(s)$, daß die Markenanzahl auf der Stelle s mit größer werdendem i streng monoton wächst. Die Stelle s und somit das Petrinetz $\mathcal{N}$ sind unbeschränkt.
„⇒“: Existieren keine zwei Markierungen $m^0, m^1 \in \mathcal{R}_\mathcal{N}(m_0)$, wie oben beschrieben, so gilt für alle Markierungen $m, m' \in \mathcal{R}_\mathcal{N}(m_0)$ mit $m \xrightarrow{*} m'$: $m' \leq m$ oder m und m' sind „unvergleichbar“ (d. h. $\exists s, s' \in S$: $m(s) > m'(s) \ \wedge \ m(s') < m'(s')$). Die Menge $\mathcal{M}(m) := \{\, m' \mid m' \in \mathcal{R}_\mathcal{N}(m) \ \wedge \ m' \leq m \,\}$ ist endlich. Die Anzahl der direkten Nachfolgemarkierungen einer Markierung m ist durch $|T|$ beschränkt und

somit ebenfalls endlich. Das Petrinetz $\mathcal{N}$ kann also nur dann unbeschränkt sein, wenn eine unendliche Markierungsfolge

$$\pi := m^0 \xrightarrow{t} m^1 \xrightarrow{t'} m^2 \xrightarrow{t''} m^3 \xrightarrow{t'''} \cdots$$

mit unendlich vielen unterschiedlichen Markierungen existiert. Angenommen, eine solche Markierungsfolge π existiert. Dann läßt sich durch die folgenden Schritte eine unendliche Markierungsfolge π' konstruieren.
Beginne mit m^0 und streiche die endlich vielen Markierungen, für die $m^i \leq m^0$ gilt. Nimm die nächste verbleibende Markierung aus π und verfahre analog. Wiederhole dies unendlich oft.
Die so konstruierte Markierungsfolge π' müßte aus unendlich vielen paarweise unvergleichbaren Markierungen bestehen. Da aber zu der Markierung m^0 nur eine endliche Anzahl von paarweise unvergleichbaren Markierungen zugefügt werden können, erhält man einen Widerspruch zur Annahme. Folglich ist das Petrinetz $\mathcal{N}$ beschränkt. □

Korollar 2
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Jede unendliche Zustandsabfolge

$$m^0 \xrightarrow{t_0} m^1 \xrightarrow{t_1} \cdots$$

mit $m^i \in \mathcal{R}_{\mathcal{N}}(m_0), i \in \mathbb{N}$, die keine zwei Zustände m^i, m^j mit $m^i \xrightarrow{*} m^j$ und $m^j > m^i$ enthält, besteht aus endlich vielen unterschiedlichen Zuständen.

Beweis: Siehe dazu den Beweis zu Satz 4. □

Dieses Beschränktheitskriterium ermöglicht es uns also, ein Verfahren anzugeben, welches durch die Betrachtung der erreichbaren Markierungen die Beschränktheit bzw. Unbeschränktheit eines Petrinetzes überprüft.
Wir müssen beim Erzeugen aller Zustände nur darauf achten, ob es einen von einem Zustand m erreichbaren Zustand m' gibt, der komponentenweise betrachtet größer als m ist.
Der Algorithmus 1 (Bounded) berechnet für ein beschränktes Petrinetz in einem Tiefendurchlaufverfahren die Menge aller vom Initialzustand aus erreichbaren Zustände. Im Falle, daß das Petrinetz unbeschränkt ist, bricht der Algorithmus nach endlich vielen Schritten ab und zeigt die Unbeschränktheit des Petrinetzes an. Der Abbruchtest benötigt für jeden neu erzeugten Zustand dessen Vorgängerzustände. Daher werden die Zustandsübergänge in der Kantenmenge $\mathcal{K}$ protokolliert. Die Funktion (TransClosure), die den transitiven Abschluß über der bis dahin schon berechneten Kantenmenge erzeugt, wird zur Bestimmung der Vorgängerzustände eines Zustandes verwendet.
Reale Systeme haben (meist) eine endliche Anzahl von Ressourcen oder lassen sich bei der Modellierung auf ein endliches System abbilden, so daß eine Beschränkung

Algorithmus 1 (Beschränktheitstest)

```
1  Bounded([S, T, F, m_0]) ≡
2    R := {m_0};
3    K := ∅;
4
5    proc DFS(m) ≡
6      forall { t | t⁻ ≤ m } do
7        m' := m + Δt;
8        K := K ∪ {[m, t, m']};
9        if m' ∉ R then
10         if ∃[m'', _, m'] ∈ TransClosure(K) ∧ (m'' < m') then
11           exit("unbeschränkt")
12         else
13           R := R ∪ {m'};
14           DFS(m')
15         fi
16       fi
17     od
18   end DFS;
19
20 begin
21   DFS(m_0);
22   exit("beschränkt")
23 end.
```

auf endliche Petrinetze keine allzu große Einschränkung bedeutet. Ein Spezialfall der beschränkten Petrinetze bilden die Petrinetze, bei denen sich auf jeder Stelle maximal eine Marke befinden kann. Sie werden als *sichere* Petrinetze bezeichnet. Da sich jedes beschränkte Petrinetz in ein korrespondierendes sicheres Petrinetz (durch Aufspaltung von beschränkten Stellen in mehrere sichere Stelle) umformen läßt, betrachten wir im folgenden (wenn dies nicht explizit anders angegeben ist) nur sichere Petrinetze.

2.4 Semantik von Petrinetzen

Jedem beschränkten Petrinetz entspricht ein endlicher Automat, dessen Zustände der Menge der erreichbaren Markierungen und dessen Zustandsübergänge dem Schalten einzelner Transitionen entsprechen. Durch die Zustände und den beschrifteten Kanten zwischen diesen kann eine Beziehung zum Petrinetz und damit zum modellierten System hergestellt werden.

Definition 13 (Schaltfolgen)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Wir bezeichnen als *endliche Schaltfolgen* von $\mathcal{N}$ alle endlichen Transitionsfolgen $\delta = t_1 t_2 \ldots t_n$ mit $m_0 \xrightarrow{\delta} m$. Analog werden als *unendliche Schaltfolgen* von $\mathcal{N}$ die Transitionsfolgen $\delta = t_1 t_2 \ldots$ mit $m_i \xrightarrow{t_{i+1}} m_{i+1}$, $i \in \mathbb{N}$ bezeichnet.

Definiert man nun das Verhalten eines Petrinetzes durch die Menge aller endlichen und unendlichen Schaltfolgen, so hat man implizit die Vorstellung eines globalen Beobachters im Hinterkopf, der zu jeder Zeit festlegt, welche Transition im Falle mehrerer nebenläufiger Transitionen zuerst schalten wird. Diese Vorstellung vom Verhalten eines Systems nennt man *Interleaving-Semantik*.
Das Interleaving-Verhalten wird normalerweise durch den Erreichbarkeitsgraph repräsentiert.

Definition 14 (Erreichbarkeitsgraph)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Als *Erreichbarkeitsgraph* von $\mathcal{N}$ bezeichnet man den Graphen $\mathcal{RG}(\mathcal{N}) := [\mathcal{R}_\mathcal{N}(m_0), \mathcal{K}_\mathcal{N}]$, der die in $\mathcal{N}$ erreichbaren Zustände als Knoten und die Menge $\mathcal{K}_\mathcal{N}$ von mit Transitionen beschrifteten Kanten hat, wobei

$$\mathcal{K}_\mathcal{N} := \{\, [m, t, m'] \mid m, m' \in \mathcal{R}_\mathcal{N}(m_0) \wedge t \in T \wedge m \xrightarrow{t} m' \,\}.$$

Das Verhalten eines Petrinetzes ist also gleichbedeutend mit der Menge aller bei m_0 beginnenden Wege durch den Erreichbarkeitsgraphen. Erweitert man den Algorithmus 1 (Bounded) in der Weise, daß bei einer Terminierung die erzeugte Zustandsmenge und die Kantenmenge ausgegeben werden, so erhält man für beschränkte Petrinetze ein Verfahren zur Konstruktion des Erreichbarkeitsgraphen.
Als Beispiel ist in Abbildung 2.2 der Erreichbarkeitsgraph des Petrinetzes in Abbildung 2.1 dargestellt. Dabei ist die Bezeichnung der Knoten so gewählt, daß anstatt des Markierungsvektors die Menge der markierten Stellen angegeben ist.

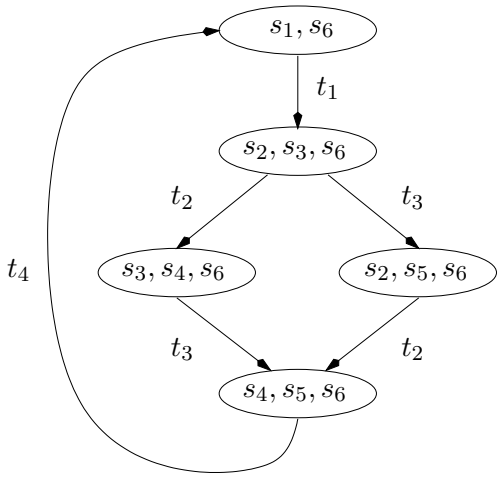


Abbildung 2.2: Erreichbarkeitsgraph

Die Erreichbarkeitsgraphen als Ausdruck des Verhaltens von Petrinetzen eignen sich zum Ablesen der (meisten) Petrinetzeigenschaften und damit der Eigenschaften der modellierten Systeme.

Zwei für die Analyse eines Systems wichtige Eigenschaften sind:

Definition 15 (tote Markierungen, Deadlocks)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Eine Markierung m von $\mathcal{N}$ heißt *tot*, wenn keine Transition $t \in T$ bei m Konzession hat. Tote Markierungen werden auch *Deadlocks* genannt.

Tote Markierungen sind Systemzustände, bei denen das System nicht mehr weiterarbeiten kann. Sie sind oft ein Anzeichen dafür, daß es Systemteile gibt, die sich gegenseitig blockieren. Bei der Überprüfung eines Systems möchte man solche Zustände natürlich erkennen und ausschließen. Tote Markierungen lassen sich im Erreichbarkeitsgraph sehr leicht bestimmen, da es sich dabei um Knoten handelt, die keine ausgehenden Kanten besitzen.

Definition 16 (reversibel)
Ein Petrinetz $\mathcal{N} = [S, T, F, m_0]$ wird *reversibel* genannt, wenn für jede Markierung $m \in \mathcal{R}_{\mathcal{N}}(m_0)$ gilt: $m \xrightarrow{*} m_0$.

In einem reversiblen Petrinetz können Transitionen, die einmal geschaltet haben, immer wieder schalten. Es gibt somit keine Systemteile, die irgendwann ihren Dienst aufgeben und damit nutzlos werden. Ein Petrinetz ist genau dann reversibel, wenn der zugehörige Erreichbarkeitsgraph stark-zusammenhängend ist.
Betrachtet man Erreichbarkeitsgraphen näher, fällt auf, daß die Information über Nebenläufigkeit (bzw. Unabhängigkeit) von Ereignissen verloren gegangen ist. Dies sind aber gerade, im Vergleich zu sequentiellen Systemen, die wichtigsten Merkmale verteilter Systeme. Entfernt man in dem Petrinetz aus Abbildung 2.1 die Stelle s_6, so sind die Transitionen t_2 und t_3 bei der Markierung $(0, 1, 1, 0, 0, \_)$ nebenläufig. Der Erreichbarkeitsgraph in Abbildung 2.2 verändert sich aber dadurch, bis auf das Verschwinden von s_6 aus den Knotenbeschriftungen, nicht.
Nebenläufige (bzw. unabhängige) Ereignisse werden durch die Interleaving-Semantik als Permutation aller möglichen Anordnungen ausgedrückt. Dies kann zu einem exponentiellen Zuwachs von Markierungen führen, was wohl den Hauptkritikpunkt des Interleaving-Ansatzes darstellt. Benutzt man Petrinetze zur Modellierung stark nebenläufiger Systeme, was ein Hauptanwendungsgebiet darstellt, stößt man mit der Konstruktion des Erreichbarkeitsgraphen sehr schnell an die Grenzen des Machbaren. Das Petrinetz in Abbildung 2.3 enthält n nebenläufige Transitionen und besitzt einen Erreichbarkeitsgraphen mit 2^n Zuständen.

2.5 Stelleninvarianten

Eigenschaften eines dynamischen Systems, die in jedem erreichbaren Zustand gültig sind, sich also auf die Verteilung von Marken auf den Stellen des Netzes beziehen,

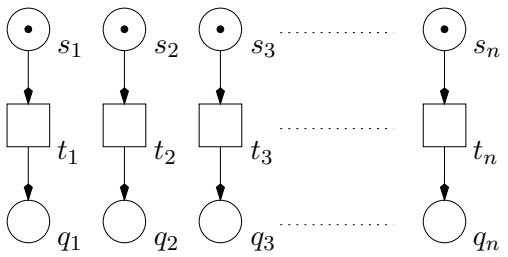


Abbildung 2.3: Exponentielles Wachstum

werden Stelleninvarianten genannt. Solche Stelleninvarianten lassen sich in einem Netz rein anhand der Netzstruktur bestimmen.

Definition 17 (Stelleninvarianten)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz mit der Inzidenzmatrix C. Jede ganzzahlige Lösung von $C^\mathsf{T} \cdot i = \vec{0}$ ist eine *Stelleninvariante* (bzw. *S-Invariante*) von $\mathcal{N}$.

Stelleninvarinaten sind also ganzzahlige Lösungen homogener Gleichungssysteme. Diese lassen sich z. B. mit dem Gaußschen Eliminationsverfahren (vgl. z. B. [Sch86]) berechnen.

Definition 18 (semipositiv, minimal)

1. Wir bezeichnen Stelleninvarianten, bei denen alle Komponenten größer oder gleich Null sind, als *semipositiv*.
2. Ist i eine semipositive Stelleninvariante, so bezeichnen wir die Menge der Stellen von i, deren Komponente positiv ist, als *Träger* von i und schreiben dafür $\langle i \rangle$.
3. Eine semipositive Stelleninvariante i heißt *minimal*, wenn es keine andere semipositive Stelleninvariante j gibt mit $\langle j \rangle \subset \langle i \rangle$.

Die fundamentale Eigenschaft von Stelleninvarianten, die Invarianz, folgt aus der Markierungsgleichung.

Satz 5 (Invarianz von Invarianten)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz mit einer Stelleninvarianten i. Dann gilt für jede von der Initialmarkierung m_0 erreichbare Markierung m:

$$i^\mathsf{T} \cdot m = i^\mathsf{T} \cdot m_0.$$

Beweis: Da $m \in \mathcal{R}_\mathcal{N}(m_0)$ existiert eine Transitionsfolge δ mit $m_0 \xrightarrow{\delta} m$. Aus der Markierungsgleichung erhalten wir $m = m_0 + C \cdot \Psi(\delta)$. Daher gilt:

$$\begin{aligned} i^\mathsf{T} \cdot m &= i^\mathsf{T} \cdot (m_0 + C \cdot \Psi(\delta)) \\ &= i^\mathsf{T} \cdot m_0 + i^\mathsf{T} \cdot C \cdot \Psi(\delta) \\ &\overset{(i^\mathsf{T} \cdot C = \vec{0}^\mathsf{T})}{=} i^\mathsf{T} \cdot m_0 \end{aligned}$$

□

Mit anderen Worten ist also die Anzahl der Marken, gewichtet mit der Stelleninvariante i, bei jeder erreichbaren Markierung konstant.

Satz 6 (Hinreichendes Kriterium für die Beschränktheit von Petrinetzen)
Es sei $\mathcal{N} = [S, T, F, m_0]$ ein Petrinetz. Existiert zu jeder Stelle $s \in S$ eine semipositive Stelleninvariante i_s von $\mathcal{N}$ mit $i_s(s) > 0$, dann ist $\mathcal{N}$ beschränkt.

Beweis: Sei $s \in S$ und i_s die semipositive Stelleninvariante mit $i_s(s) > 0$. Des weiteren sei $m \in \mathcal{R}_{\mathcal{N}}(m_0)$. Dann gilt:

$$i_s^{\mathsf{T}} \cdot m_0 = i_s^{\mathsf{T}} \cdot m \geq i_s(s) \cdot m(s) \geq m(s).$$

Die Markenzahl auf der Stelle s in jeder von m_0 erreichbaren Markierung ist durch $i_s^{\mathsf{T}} \cdot m_0 \in \mathbb{N}$ beschränkt. Da diese Folgerung für jede Stelle gültig ist, ist $\mathcal{N}$ beschränkt.

□

Dieser Satz liefert uns ein strukturelles Kriterium zur Bestimmung der Beschränktheit von Petrinetzen. Leider handelt es sich hierbei nur um ein hinreichendes Kriterium, da z. B. das Petrinetz, bestehend aus einer markierten Stelle s, einer Transition t und einer Kante zwischen der Stelle und der Transition beschränkt ist, aber keine semipositive Stelleninvariante i mit $i(s) > 0$ besitzt.
Im folgenden wollen wir eine spezielle Klasse von Stelleninvarianten, die 1-Stelleninvarianten, näher betrachten.

Definition 19 (1-Stelleninvarianten)
Eine minimale, semipositive Stelleninvariante i eines Petrinetzes $\mathcal{N}$ bezeichnen wir als *1-Stelleninvariante* (bzw. *1-S-Invariante*), falls gilt: $i^{\mathsf{T}} \cdot m_0 = 1$.

Eine 1-Stelleninvariante definiert also eine Stellenmenge $S_i := \{\, s \mid i(s) = 1 \,\}$, so daß bei jeder von m_0 aus erreichbaren Markierung immer genau eine Stelle von S_i markiert ist. Für jede in der Anfangsmarkierung markierte Stelle s kann man durch die Modifikation des Invariantengleichungssystems $C^{\mathsf{T}} \cdot i = \vec{0}$ alle 1-Stelleninvarianten berechnen, in deren Träger s enthalten ist. Dazu initialisiert man Teile des Vektors i mit vorgegebenen Werten. Die Position der Stelle s wird mit 1 und alle Positionen der Stellen $q \neq s$, die bei m_0 markiert sind, werden mit 0 markiert. Alle sich mit dem Gaußschen Eliminationsverfahren ergebenden Lösungen sind die der Stelle s zugehörigen 1-Stelleninvarianten.
Die 1-Stelleninvarianten des Petrinetzes aus Abbildung 2.1 sind die Stellenmengen $\{\, s_6 \,\}$, $\{\, s_1, s_2, s_4 \,\}$ und $\{\, s_1, s_3, s_5 \,\}$.
Die Berechnung der 1-Stelleninvarianten läßt sich oft dadurch vereinfachen, daß man vor der Lösung des homogenen Gleichungssystems die Größe der Petrinetze durch naheliegende Transformationen strukturell verkleinert. Dazu führt man sogenannte *Makrostellen* (bzw. *Makrotransitionen*) ein, die Stellen (bzw. Transitionen) verschmelzen.

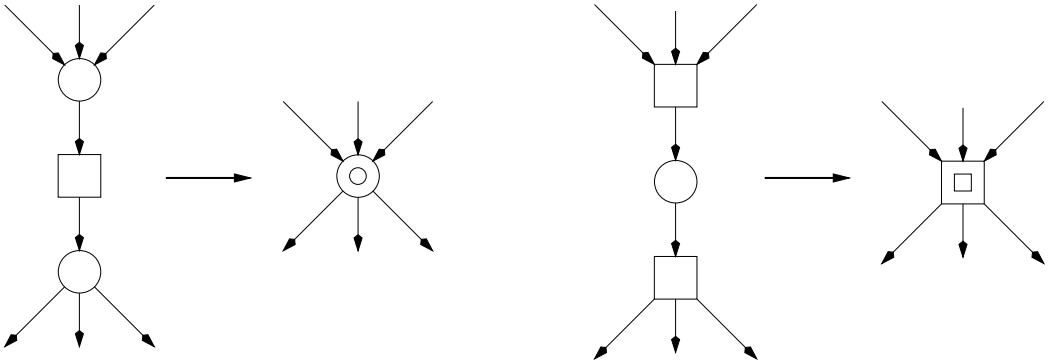


Abbildung 2.4: Reduktionsregeln

Nach der Berechnung der 1-Stelleninvarianten im reduzierten Petrinetz muß man diese passend erweitern. Jede in einer 1-Stelleninvarianten vorkommende Makrostelle wird durch die beiden ursprünglichen Stellen ersetzt. Einer 1-Stelleninvarianten, die eine Stelle aus dem Vor- und dem Nachbereich einer Makrotransition enthält, wird die der Makrotransition zugeordnete Stelle zugefügt.
Im folgenden betrachten wir nur noch minimale 1-Stelleninvarianten und nennen sie kurz Stelleninvarianten.

2.6 Zusammenfassende Bemerkungen

Petrinetze sind eine adäquate Beschreibungssprache nebenläufiger Systeme. Als Vorteil erweist sich die mathematische Fundierung des Formalismus. Sie eröffnet uns eine Vielzahl von Ansätzen zur Analyse von Systemeigenschaften. Betrachtet man die Interleaving-Semantik, so ist das Verhalten eines endlichen Petrinetzes durch seinen Erreichbarkeitsgraphen repräsentiert. Er erweist sich daher (beinahe) als ein Universalmittel zu Bestimmung von Petrinetzeigenschaften. Leider kann der Erreichbarkeitsgraph, selbst für relativ kleine Petrinetze, schnell sehr groß werden.
Eine Möglichkeit der Handhabung großer Zustandsmengen stellen die im nächsten Kapitel betrachteten binären Entscheidungsgraphen dar. Zur Optimierung des Zusammenspiels zwischen Petrinetzen und binären Entscheidungsgraphen kann man auf strukturelle Eigenschaften der Petrinetze zurückgreifen.
Die in diesem Kapitel dafür speziell entwickelten 1-S-Invarianten werden wir im Kapitel 4 bei der Zusammenführung von Petrinetzen und binären Entscheidungsgraphen näher betrachten.

3 Binäre Entscheidungsgraphen

Viele Probleme der diskreten Mathematik und der Informatik lassen sich auf die Manipulation von Objekten über endlichen Grundmengen zurückführen. Durch eine binäre Codierung der Elemente der Grundmenge lassen sich alle endlichen Probleme vollständig mit Booleschen Funktionen beschreiben. Man benötigt daher eine kompakte Darstellung und effiziente Möglichkeiten zur Manipulation Boolescher Formeln.

Seit einigen Jahren haben sich dazu die binären Entscheidungsgraphen, kurz BDDs (*binary decision diagrams*), etabliert. Die Grundidee basiert auf der Shannon-Entwicklung Boolescher Funktionen, die Boolesche Funktionen als eine Kaskade von binären Entscheidungen bezüglich der einzelnen Variablenbelegungen darstellt.

Binäre Entscheidungsgraphen wurden schon 1959 von C. Y. Lee [Lee59] und später von S. B. Akers [Ake78] als Datenstruktur für Boolesche Funktionen vorgeschlagen. Der Durchbruch gelang aber erst 1986, als R. Bryant [Bry86, Bry92] durch Hinzunahme von Ordnungsbedingungen sowie Reduktionsmechanismen den Ansatz entscheidend verbesserte. Diese sogenannten reduzierten geordneten binären Entscheidungsgraphen, kurz ROBDDs (*reduced ordered binary decision diagrams*), besitzen Eigenschaften, die eine effiziente rechnergestützten Behandlung großer Booleschen Funktionen ermöglichen. Durch die eingeführten Restriktionen erhält man für jede Boolesche Funktion eine kanonische Darstellung, so daß Äquivalenztests zwischen Booleschen Funktionen sehr effizient möglich sind.

Wie alle anderen möglichen Repräsentationen von Booleschen Funktionen besitzen auch die ROBDDs die Eigenschaft, daß fast alle Booleschen Funktionen eine exponentielle Darstellungsgröße erfordern. Glücklicherweise hat sich aber gezeigt, daß viele in der Praxis relevante Boolesche Funktionen sich sehr kompakt mit ROBDDs darstellen lassen.

Wie wir im vorherigen Kapitel gesehen haben, beschreibt ein Erreichbarkeitsgraph das vollständige Interleaving-Verhalten eines Petrinetzes. Er eignet sich also als Ausgangspunkt für die Analyse des Petrinetzverhaltens. Als Hauptproblem dabei erweist sich allerdings das starke Wachstum der Anzahl der Zustände.

Es zeigt sich, daß eine enge Beziehung zwischen Mengen und Boolesche Funktionen existiert. Daher eignen sich binäre Entscheidungsgraphen als effiziente Datenstruktur zur Speicherung und Manipulation von großen Mengen von Zuständen.

Einen guten Überblick über binäre Entscheidungsgraphen und deren Handhabung bie-

ten, neben den Arbeiten von R. Bryant [Bry86, Bry92], die Standardwerke [And97, DB98, MT98].

3.1 Boolesche Algebren

Im Jahre 1938 zeigte C. E. Shannon [Sha38], wie man die grundlegenden Regeln der klassischen Logik und der elementaren Mengentheorie auch für die Analyse und Beschreibung von digitalen Schaltkreisen (durch Schaltfunktionen) verwenden kann. Dadurch wurde der Bogen zwischen Theorie und Praxis gespannt.

3.1.1 Boolesche Algebra

Definition 20 (Boolesche Algebra)
Eine *Boolesche Algebra* ist ein Tupel $\mathcal{B} = (A, +, \cdot, \bar{}, 0, 1)$, bestehend aus einer *Trägermenge* A, die zwei ausgezeichnete Elemente 0 (*Nullelement*) und 1 (*Einselement*) enthält, einer einstelligen Operation $\bar{}$ und zwei binären Operationen $+, \cdot$ auf A, falls für alle $a, b, c \in A$ folgendes gilt:

Kommutativität: $a + b = b + a$ und $a \cdot b = b \cdot a$.

Distributivität: $a + (b \cdot c) = (a + b) \cdot (a + c)$ und $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Identität: $a + 0 = a$ und $a \cdot 1 = a$.

Komplement: $a + \bar{a} = 1$ und $a \cdot \bar{a} = 0$.

Das Element $\bar{a} \in A$ wird als *Komplement* von a bezeichnet. Im folgenden betrachten wir nur Boolesche Algebren mit endlicher Trägermenge. Des weiteren wollen wir vereinbaren, daß die Operation $\bar{}$ stärker als $\cdot$ und die Operation $\cdot$ stärker als $+$ bindet. Oft lassen wir auch $\cdot$ zwischen den Argumenten einfach weg, so daß sich aus $a \cdot b$ die kürzere Schreibweise ab ergibt.

Definition 1 (Mengenalgebra)
Das Tupel $(2^S, \cup, \cap, \bar{}, \emptyset, S)$ ist eine (endliche) Boolesche Algebra und wird als (endliche) *Mengenalgebra* bezeichnet. Hierbei bezeichnen 2^S die Potenzmenge einer endlichen Menge S und $\cup, \cap$ die elementaren Mengenoperationen. Das Komplement eines Elementes $X \subseteq S$ ergibt sich als $S \setminus X$.

Die bestehende enge Beziehung zwischen Booleschen Algebren und den Mengenalgebren verdeutlicht der folgende Satz.

Satz 7 (Repräsentationssatz von Stone [Sto36])
Jede endliche Boolesche Algebra ist isomorph zur Mengenalgebra einer endlichen Menge.

Der Satz von Stone besagt, daß jede endliche Boolesche Algebra isomorph zu einer endlichen Mengenalgebra ist. Daraus folgt, daß alle Booleschen Algebren, deren Träger die Kardinalität n haben, isomorph zu der Mengenalgebra sind, deren Träger ebenfalls die Kardinalität n hat. Wegen der Isomorphie kann es also keine endlichen Booleschen Algebren mit einer beliebigen Anzahl von Trägerelementen geben, vielmehr muß die Anzahl der Trägerelemente der Anzahl der Elemente der Potenzmenge einer endlichen Menge entsprechen. Daher ergeben sich als Kardinalität von Trägermengen nur Zweierpotenzen.

Beispiel 1 (Isomorphe Booleschen Algebren)

1. $(\{\mathsf{f},\mathsf{t}\}, \vee, \wedge, \neg, \mathsf{f}, \mathsf{t})$
2. $(\{\emptyset, S\}, \cup, \cap, ^{-}, \emptyset, S)$
3. $(\{0,1\}, max\{a,b\}, min\{a,b\}, \{0 \leftarrow 1, 1 \leftarrow 0\}, 0, 1)$

Die „einfachste“ Boolesche Algebra hat eine Trägermenge mit nur zwei Elementen. Sie wird durch ihre Verwandtschaft zu digitalen Schaltsystemen als *Schaltalgebra* bezeichnet.
Auf Grund des Satzes von Stone ist es also für das Arbeiten mit der Schaltalgebra nicht relevant, ob wir Mengenoperationen, logischen Operationen oder Max-/Minimums-Operationen betrachten.

3.1.2 Boolesche Ausdrücke und Funktionen

Definition 21 (Boolescher Ausdruck)
Sei $\mathcal{B} = (A, +, \cdot, ^{-}, 0, 1)$ eine Boolesche Algebra. Ein Ausdruck bestehend aus n Variablensymbolen $x_1, \ldots, x_n$, den Symbolen $+, \cdot, ^{-}$ und den Elementen aus A heißt n-stelliger *Boolescher Ausdruck*, falls er durch endlich viele Anwendungen der folgenden rekursiven Regeln erzeugt werden kann.

1. Die Elemente von A sind Boolesche Ausdrücke.
2. Die Variablensymbole $x_1, \ldots, x_n$ sind Boolesche Ausdrücke.
3. Sind G und H Boolesche Ausdrücke so auch $(G + H)$, $(G \cdot H)$ und $\overline{G}$.

Jeder Boolesche Ausdruck G induziert eine *Boolesche Funktion* $f_G\colon A^n \to A$, $(a_1, \ldots, a_n) \mapsto f_G(a_1, \ldots, a_n)$. Hierbei bezeichnet $f_G(a_1, \ldots, a_n)$ das Element von A, welches sich durch Ersetzen der Variablensymbole x_i von G durch die Elemente $a_i \in A$ und der anschließenden Auswertung der Booleschen Operatoren und der Komplementbildung, unter Berücksichtigung der Klammerung, ergibt.

Definition 22 (Boolesche Funktionen)
Sei $\mathcal{B} = (A, +, \cdot, \bar{}, 0, 1)$ eine Boolesche Algebra. Eine n-stellige Funktion $f\colon A^n \to A$ wird *Boolesche Funktion* genannt, wenn sie von einem Booleschen Ausdruck induziert wird.

Die Booleschen Funktionen über der Schaltalgebra werden als *Schaltfunktionen* bezeichnet. Der Schaltalgebra kommt dabei, wie folgender Satz verdeutlicht, eine Sonderrolle zu.

Satz 8
Besitzt die Trägermenge einer Booleschen Algebra $\mathcal{B} = (A, +, \cdot, \bar{}, 0, 1)$ genau zwei Elemente, dann ist jede Funktion $f\colon A^n \longrightarrow A$ auch eine Boolesche Funktion.

Beweis: Siehe [MT98]. □

Dies gilt nicht für Trägermengen mit größerer Kardinalität. Jede Funktion auf einer Booleschen Algebra mit zwei Elementen als Träger ist also eine Schaltfunktion. Jeder Boolesche Ausdruck über der Schaltalgebra induziert eine eindeutige Boolesche Funktion. Allerdings ist die Beziehung zwischen den Booleschen Ausdrücken und den Booleschen Funktionen nicht injektiv, d. h. es gibt mehrere verschiedene Boolesche Ausdrücke, die die gleiche Boolesche Funktion induzieren. Wie wir z. B. schon aus der Definition einer Booleschen Algebra wissen, gilt $a + (b \cdot c) = (a + b) \cdot (a + c)$.
Die Menge der n-stelligen Schaltfunktionen entspricht der Menge aller Abbildungen einer 2^n-elementigen Menge in eine zweielementige Menge. Es gibt also $2^{(2^n)}$ verschiedene Schaltfunktionen mit n Variablen.
Die Menge der n-stelligen Schaltfunktionen als Trägermenge und die entsprechende Anpassung der Operationen ergeben wiederum eine (endliche) Boolesche Algebra.

Satz 9
Sei $\mathcal{B} = (A, +, \cdot, \bar{}, 0, 1)$ eine Boolesche Algebra. Die Menge $\mathcal{F}_n$ aller n-stelligen Booleschen Funktionen über der Booleschen Algebra $\mathcal{B}$ bildet eine Boolesche Algebra $(\mathcal{F}_n, +, \cdot, \bar{}, 0, 1)$ bezüglich der folgenden Operationen und Elemente:

$$\begin{aligned}
(f+g)(a_1,\ldots,a_n) &:= f(a_1,\ldots,a_n) + g(a_1,\ldots,a_n)\\
(f\cdot g)(a_1,\ldots,a_n) &:= \overline{f(a_1,\ldots,a_n)}\cdot g(a_1,\ldots,a_n)\\
(\bar{f})(a_1,\ldots,a_n) &:= \overline{f(a_1,\ldots,a_n)}\\
(\mathbf{0})(a_1,\ldots,a_n) &:= 0\\
(\mathbf{1})(a_1,\ldots,a_n) &:= 1
\end{aligned}$$

Beweis: Durch einfaches Überprüfen der Definition einer Booleschen Algebra. □

Im folgenden wollen wir uns auf die Betrachtung der für uns interessanten Schaltalgebra und deren n-stelligen Schaltfunktionen beschränken. Wir verwenden für die

Schaltalgebra die Darstellung $\mathcal{B} = (\mathbb{B}, +, \cdot, ^-, 0, 1)$. Die Trägermenge $\mathbb{B}$ definieren wir als die Menge $\{0, 1\}$. Des weiteren interpretieren wir 0 und 1 als die Wahrheitswerte *falsch* und *wahr* und $+, \cdot, ^-$ als $\vee$ (*logisches oder*), $\wedge$ (*logisches und*) und $\neg$ (*logische Negation*).
Wie wir gesehen haben, existieren verschiedene isomorphe Darstellungen, so daß wir von Fall zu Fall zwischen ihnen wechseln werden.
Wir betrachten nun den Zusammenhang zwischen Schaltfunktionen und Mengen näher.
Jeder Menge $S \subseteq \mathbb{B}^n$ kann durch ihre *charakteristische Funktion* χ_S von S

$$\chi_S(x_1, \ldots, x_n) = 1 \Longleftrightarrow (x_1, \ldots, x_n) \in S,$$

eindeutig eine n-stellige Schaltfunktion zugeordnet werden.

Definition 23 (Erfüllende Belegung)
Sei f eine n-stellige Schaltfunktion.

1. Wir bezeichnen einen Vektor $a = (a_1, \ldots, a_n) \in \mathbb{B}^n$ als *erfüllende Belegung* der Schaltfunktion f, falls $f(a) = 1$ gilt.

2. Die Menge aller erfüllenden Belegungen von f wird mit $on(f)$ bezeichnet und ist wie folgt definiert:

$$on(f) := \{\, (a_1, \ldots, a_n) \in \mathbb{B}^n \mid f(a_1, \ldots, a_n) = 1 \,\}$$

Da die Menge aller erfüllenden Belegungen einer Schaltfunktion f diese eindeutig beschreibt, kann man f mit $on(f)$ identifizieren.
Genauer gilt die Beziehung $f = \chi_{on(f)}$.
Folglich können Mengendarstellungen und Mengenoperationen vollständig auf die Manipulation von Schaltfunktionen zurückgeführt werden. Dies weist uns die Richtung zur Handhabung von Zustandsmengen als Schaltfunktionen, was wir später näher betrachten werden.

3.1.3 Shannon-Entwicklung

Grundlage für das Arbeiten mit Schaltfunktionen ist die Shannon-Entwicklung. Sie beschreibt den Zusammenhang zwischen einer Schaltfunktion f und ihren Cofaktoren. Cofaktoren sind Unterfunktionen einer Schaltfunktion f, die aus f durch Belegen einer Eingangsvariablen mit einer Konstanten entstehen.

Definition 24 (Cofaktoren)
Sei f eine n-stellige Schaltfunktion. Die Unterfunktion $f_{x_i} := f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ wird als *positiver Cofaktor* von f bezüglich x_i bezeichnet. Analog wird die Unterfunktion $f_{\overline{x_i}} := f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ als *negativer Cofaktor* von f bezüglich x_i bezeichnet.

Man bezeichnet eine Boolesche Variable x_i oder deren Komplement als *Literal* (zur Variablen x_i). Die Cofaktorbildung läßt sich auf bestimmte Mengen von Literalen, die sogenannten *Monome*, ausweiten.

Definition 25 (Monome)
Sei V einen Menge von Booleschen Variablen. Ein *Monom* m ist eine Menge von Literalen zu V, so daß gilt: Falls $x_i \in V$ dann $\overline{x_i} \notin V$ und umgekehrt.

Der Cofaktor von f bzgl. des Monoms $m = \{l_i\} \cup m'$, wobei $l_i \notin m'$ für x_i oder $\overline{x_i}$ steht, definiert man rekursiv als $f_m := [f_{l_i}]_{m'}$.

Satz 10 (Shannon-Entwicklung)
Sei f eine n-stellige Schaltfunktion, dann gilt:

$$f = \overline{x_i} \cdot f_{\overline{x_i}} + x_i \cdot f_{x_i}, \quad \forall i \; 1 \leq i \leq n.$$

Beweis: Die Korrektheit des Satzes ergibt sich leicht durch Nachrechnen. □

Oft wird für die Shannon-Entwicklung auch der Begriff *Shannon-Zerlegung* verwendet, da de facto eine Aufspaltung der Funktion in zwei Unterfunktionen erfolgt.
Weitere wichtige Funktionen im Umgang mit Schaltfunktionen stellen die Quantifizierungen dar. Sie entsprechen den Quantifizierungen von Prädikaten in der Aussagenlogik.

Definition 26 (Quantifizierung)
Sei f eine n-stellige Schaltfunktion. Für f ist die *existentielle Quantifizierung* der Variablen x_i durch

$$\exists_{x_i} f := f_{x_i} + f_{\overline{x_i}}$$

und die *universelle Quantifizierung* durch

$$\forall_{x_i} f := f_{x_i} \cdot f_{\overline{x_i}}$$

definiert.

Die Quantifizierungen $\exists_{x_i} f$ und $\forall_{x_i} f$ sind wieder Schaltfunktionen, die allerdings nicht mehr von der Variablen x_i abhängig sind. Aus diesem Grund wird die Quantifizierung oft auch als Abstraktion bezeichnet. Die existentielle Quantifizierung von f ist die kleinste von x_i unabhängige Schaltfunktion mit $on(f) \subseteq on(\exists_{x_i} f)$. Die universelle Quantifizierung von f ist die größte von x_i unabhängige Schaltfunktion mit $on(\forall_{x_i} f) \subseteq on(f)$.

Beispiel 2 ($f = x_2x_3 + x_1\overline{x_2x_3} + \overline{x_1}x_3$)
positiver Cofactor: $f_{x_1} = x_2x_3 + \overline{x_2x_3}$

negativer Cofactor:	$f_{\overline{x_1}} = x_3$
existentielle Quantifizierung:	$\exists_{x_1} f = \overline{x_2} + x_3$
universelle Quantifizierung:	$\forall_{x_1} f = x_2 x_3$

Die existentielle Quantifizierung $\exists_{x_1} f$ entspricht der Schaltfunktion, deren Funktionswert für alle die Belegungen von x_2 und x_3 gleich 1 ist, für die eine Belegung von x_1 in f existiert, so daß f ebenfalls den Funktionswert 1 hat. Die universelle Quantifizierung $\forall_{x_1} f$ entspricht der Schaltfunktion, deren Funktionswert für alle die Belegungen von x_2 und x_3 gleich 1 ist, für die f unter allen Belegungen von x_0 den Funktionswert 1 hat.

3.1.4 Darstellungsformen Boolescher Ausdrücke

Zum Umgang mit Booleschen Funktionen benötigt man eine vollständige und möglichst eindeutige Beschreibung dieser Funktionen.
Die klassischen Darstellungsformen basieren im wesentlichen auf den folgenden Ideen:

- Darstellung durch systematische Tabellierung der Funktionswerte (z. B. Wahrheitstabellen).
- Darstellung durch Angabe einer Methode zur Berechnung der Funktion (z. B. Boolesche Ausdrücke, disjunktive oder konjunktive Normalform).
- Darstellung durch Beschreibung eines Schemas zur Auswertung der Funktion (z. B. Entscheidungsbäume).

Für eine rechnergestützte Anwendung Boolescher Funktionen benötigt man möglichst noch zusätzlich eine Darstellungsform mit den folgenden Eigenschaften:

- Kompaktheit der Darstellung,
- effiziente Auswertung und Bearbeitung der Booleschen Funktionen,
- effizienter Äquivalenztest.

Leider ist es schon durch die große Anzahl $2^{(2^n)}$ verschiedener n-stelliger Boolescher Funktionen nicht möglich, eine Darstellungsform zu finden, für die alle geforderten Eigenschaften für alle Funktionen erfüllt sind. Würde man für jede Funktion nur eine eindeutige Zahl zur Codierung verwenden, hätte diese schon eine Darstellung mit exponentieller Länge. Als Ausweg bleibt daher, eine Darstellungsform zu finden, die für ein spezielle Problemklasse oder eine spezielle Anforderung einen guten Kompromiß zwischen den sich „widersprechenden“ Zielen Kompaktheit und Effizienz bietet.

3.2 Binäre Entscheidungsgraphen

Eine Möglichkeit, Schaltfunktionen darzustellen, basiert auf der Beschreibung eines Schemas zur Auswertung der Funktionen. Dabei wird die Zuordnung eines Funktionswertes zu einer Belegung der Eingangsvariablen durch eine Folge von Tests (Entscheidungen) bestimmt.
Bei jedem Test erfolgt eine Unterteilung der Menge der Belegungen bezüglich des Wertes einer Eingangsvariablen. Diese Auswertungsstrategie läßt sich mittels eines sogenannten *binären Entscheidungsgraphen*, kurz BDD (*binary decision diagram*), darstellen.

Definition 27 (Binärer Entscheidungsgraph)
Ein *binärer Entscheidungsgraph (BDD)* auf den Variablen $X = \{ x_1, \ldots, x_n \}$ ist ein Tupel $[V, E, var, v_0]$ mit:

1. V ist eine endliche Knotenmenge.
2. $E \subseteq V \times \mathbb{B} \times V$ ist eine endliche Menge von Kanten zwischen den Knoten, die mit 0 bzw. 1 beschriftet sind.
3. $v_0 \in V$.
4. $[V, E, v_0]$ bildet einen gerichteten azyklischen Graphen mit Wurzel v_0.
5. Die Abbildung $var\colon V \longrightarrow X \cup \mathbb{B}$ ist eine Knotenbeschriftung.

Es gelten weiter die Bedingungen:

1. Die Knotenmenge V enthält zwei *terminale Knoten*, d. h. Knoten ohne ausgehende Kanten, die mit den Konstanten 0 bzw. 1 beschriftet sind.
2. Die restlichen Knoten werden *nichtterminale Knoten* genannt und jeder nichtterminale Knoten v ist mit einer Variablen $var(v) = x_i$ beschriftet. Jeder nichtterminale Knoten besitzt genau zwei ausgehende Kanten, die mit 0 (0-*Kante*) bzw. 1 (1-*Kante*) beschriftet sind. Der durch die 0-Kante festgelegte Nachfolgeknoten wird mit $low(v)$ bezeichnet, der durch die 1-Kante bestimmte Nachfolgeknoten mit $high(v)$.
3. Auf jedem Pfad von der Wurzel v_0 zu einem terminalen Knoten kommt jede Variable x_i nur einmal vor.

Die beiden von einem nichtterminalen Knoten x_i ausgehenden Kanten entsprechen der Variablenbelegung $x_i = 0$ bzw. $x_i = 1$. Jeder nichtterminale Knoten repräsentiert also eine binäre Entscheidung.

Jede Belegung der Eingangsvariablen $x_1, \ldots, x_n$ der durch den binären Entscheidungsgraphen beschriebenen n-stelligen Schaltfunktion definiert einen eindeutigen Pfad von der Wurzel des Baumes zu einem terminalen Knoten. Die Markierung des terminalen Knotens definiert dabei den Funktionswert der Funktion bzgl. der Eingangsvariablenbelegung.
Aus einer Schaltfunktion läßt sich durch iterative Anwendung der Shannon-Entwicklung ein binärer Entscheidungsgraph gewinnen. Repräsentiert ein an einem mit x_i beschrifteten Knoten wurzelnder Entscheidungsgraph die Funktion $f(x_1, \ldots, x_n)$, dann repräsentiert der an der 0-Kante hängende Untergraph den negativen Cofaktor von f bzgl. x_i und der an der 1-Kante hängende Untergraph den positiven Cofaktor von f bzgl. x_i.
Ein binärer Entscheidungsgraph ist also ein Art „graphische Protokollierung“ einer sukzessiven Shannon-Entwicklung.

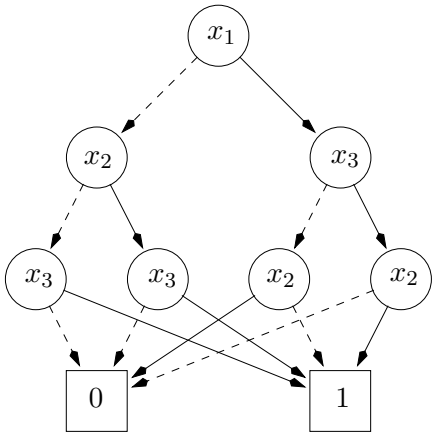


Abbildung 3.1: Binärer Entscheidungsgraph für $f = x_2x_3 + x_1\overline{x_2x_3} + \overline{x_1}x_3$.

In Abbildungen stellen wir eine 0-Kante als gestrichelten Pfeil und eine 1-Kante als durchgezogenen Pfeil dar (vgl. z. B. Abb. 3.1).

3.2.1 Geordnete binäre Entscheidungsgraphen

Binäre Entscheidungsgraphen wurden schon 1959 von C. Y. Lee [Lee59] und später von S. B. Akers [Ake78] als Datenstruktur für Boolesche Funktionen näher studiert. Erst 1986 gelang es R. Bryant [Bry86, Bry92], durch Hinzunahme von zusätzlichen Ordnungsbedingungen sowie Reduktionsmechanismen den Ansatz entscheidend zu verbessern.
Seither werden *geordnete binäre Entscheidungsgraphen*, kurz OBDDs (*ordered binary decision diagrams*), in vielen Gebieten der Informatik benutzt, um rechnergestützt Probleme, die sich mit Schaltfunktionen ausdrücken lassen, zu bearbeiten.

Definition 28 (Geordnete binäre Entscheidungsgraphen)
Sei π eine totale Ordnung auf den Variablen $X = \{ x_1, \ldots, x_n \}$. Ein *geordneter binärer Entscheidungsgraph (OBDD)* bezüglich der Variablenordnung π ist ein binärer Entscheidungsgraph $[V, E, var, v_0]$, bei dem die Knoten entlang eines Pfades bezüglich ihrer Beschriftung geordnet sind. Für jede Kante $[v, \_, v'] \in E$ gilt daher: $var(v) <_\pi var(v')$. Die Reihenfolge, in der die Variablen bei einem Durchlauf des Graphen auftreten, ist also konsistent mit der Variablenordnung π.

Ein OBDD repräsentiert eine n-stellige Schaltfunktion dadurch, daß er für jede Zuweisung an die Eingangsvariablen x_i einen eindeutig bestimmten Pfad von der Wurzel zu einem terminalen Knoten definiert. Die Beschriftung des terminalen Knotens gibt dabei den Wert der Funktion bei dieser Eingabe an.

3.2.2 Reduzierte geordnete binäre Entscheidungsgraphen

Geordnete binäre Entscheidungsgraphen repräsentieren Schaltfunktionen nicht eindeutig (vgl. Abbildung 3.2). Sie stellen also keine kanonische Repräsentationsform dar. Die Ursache dafür liegt in den folgenden Redundanzen innerhalb der Entscheidungsgraphen:

1. Der 0- und der 1-Nachfolgeknoten eines Knotens v können identisch sein. In diesem Fall liefert die „Entscheidung“, die im Knoten v getroffen wird, keine neue Information.

2. In einem Entscheidungsgraphen können Teilgraphen mehrfach auftreten. Dadurch wird die gleiche Information mehrfach repräsentiert.

Bryant hat sich in seinen Arbeiten [Bry86, Bry92] mit diesen Redundanzen beschäftigt und daraufhin das Konzept der *reduzierten binären Entscheidungsgraphen*, kurz ROBDDs (*reduced ordered binary decision diagrams*), entwickelt. Mit den ROBDDs erhielt er eine kanonische Repräsentationsform für Schaltfunktionen.
Wir wollen uns näher mit den reduzierten OBDDs beschäftigen und benötigen daher eine präzise Definition der Redundanz.

Definition 29 (Isomorphie)
Seien F und G zwei OBDDs. F und G heißen *isomorph*, wenn es eine bijektive Abbildung ϕ von der Knotenmenge von F auf die Knotenmenge von G gibt, so daß für jeden Knoten gilt:

1. Entweder sind die beiden Knoten v und $\phi(v)$ teminale Knoten mit der gleichen Markierung oder

2. es gilt: $var(v) = var(\phi(v))$, $\phi(high(v)) = high(\phi(v))$ und $\phi(low(v)) = low(\phi(v))$.

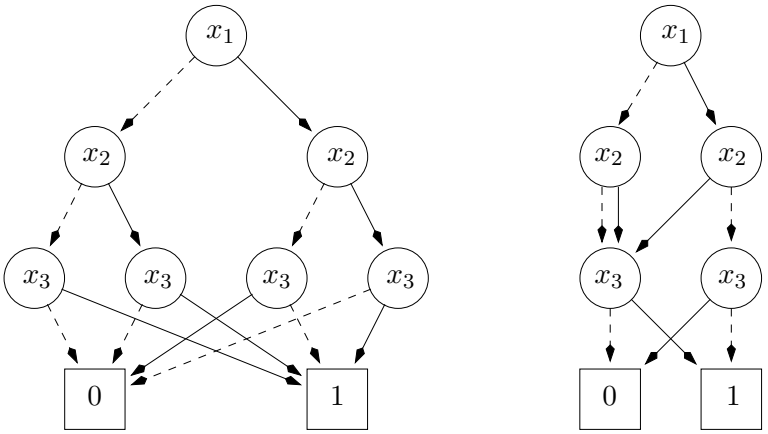


Abbildung 3.2: Zwei OBDDs der Funktion $f = x_2x_3 + x_1\overline{x_2x_3} + \overline{x_1}x_3$ bezüglich der Variablenordnung $x_1 < x_2 < x_3$.

Zwei OBDDs sind also genau dann isomorph, wenn sie als knoten- und kantenmarkierte Graphen isomorph sind.

Definition 30 (Reduziertheit)
Ein OBDD heißt *reduziert*, wenn es

1. keinen Knoten v mit $high(v) = low(v)$ und
2. kein Paar von Knoten v, w gibt, so daß die in v und w wurzelnden Unter-OBDDs isomorph sind.

Die so definierte globale Eigenschaft des Graphen kann man auch durch lokale Eigenschaften ausdrücken. Diese lassen sich durch Reduktionsregeln ausdrücken, die solange angewendet werden, bis der entsprechende OBDD vollständig reduziert ist.

Definition 31 (Reduktionsregeln)
Eliminationsregel: Wenn die 0- und die 1-Kante eines Knotens v auf den gleichen Knoten w zeigen, dann eliminiere v und lenke alle in v eingehenden Kanten auf w um.

Isomorphieregel: Wenn die nichtterminalen Knoten v und w mit der gleichen Variablen markiert sind und ihre 0-Kanten bzw. ihre 1-Kanten jeweils zum gleichen Nachfolger führen, dann eliminiere einen der beiden Knoten v, w und lenke alle in diesen Knoten eingehende Kanten auf den verbleibenden anderen Knoten um.

Den Zusammenhang zwischen den lokalen Reduktionsregeln und der globalen Grapheigenschaft der Reduziertheit liefert uns folgender Satz.

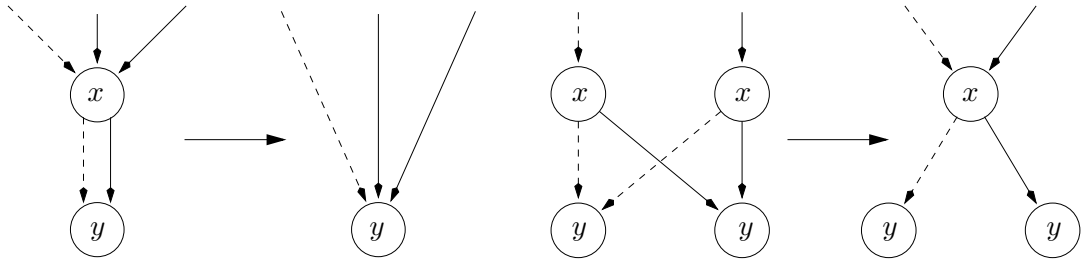


Abbildung 3.3: Reduktionsregeln

Satz 11 (Minimalität)
Ein OBDD ist genau dann reduziert, wenn keine der beiden Reduktionsregeln mehr anwendbar ist.

Beweis: Siehe [MT98]. □

Die iterative Anwendung der Reduktionsregeln sagt noch nichts darüber aus, ob so erzeugte reduzierte OBDDs eindeutig bestimmt sind, d. h. ob reduzierte OBDDs eine kanonische Darstellungsform für Schaltfunktionen darstellen. Dies liefert uns der folgende Satz.

Satz 12 (Kanonizität)
Für jede Variablenordnung π ist der reduzierte OBDD einer Schaltfunktion f, bis auf Isomorphie, eindeutig bestimmt.

Beweis: Siehe [MT98]. □

Aus den Reduktionsregeln läßt sich ein einfacher Algorithmus ableiten, der einen OBDD in einen die gleiche Schaltfunktion repräsentierenden ROBDD überführt. Der Algorithmus besteht dabei aus einer wiederholten Anwendung der Reduktionsregeln, und zwar solange, bis sich keine der Regeln mehr anwenden läßt. Bei jeder Anwendung einer Reduktionsregel verringert sich die Größe des OBDD um mindestens einen Knoten. Daraus und aus der Kanonizität der ROBDDs folgt, daß bezüglich einer festgelegten Variablenordnung π der reduzierte OBDD einer Schaltfunktion der kleinste OBDD ist, der die Schaltfunktion repräsentiert.

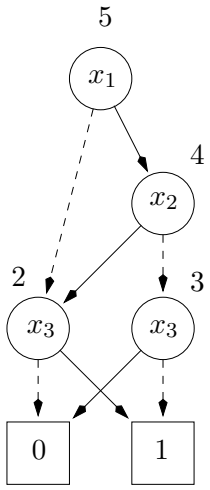
3.2.3 Konstruktion von ROBDDs

Die Shannon-Zerlegung liefert uns, wie im Abschnitt 3.2 geschildert, ein einfaches rekursives Verfahren zur Erzeugung eines OBDDs aus einem Booleschen Ausdruck. Nachträglich lassen sich nun unsere Reduktionsregeln anwenden, und man erhält einen reduzierten OBDD. Dieser Weg erscheint etwas umständlich und aufwendig. Ein anderer Ansatz versucht daher schon bei der Konstruktion darauf zu achten, daß alle

auftretenden OBDDs in reduzierter Form gehalten werden. Jede Unterfunktion darf also nur einmal repräsentiert werden.
Bei jedem zu erzeugenden Knoten v mit der Variablenbeschriftung $var(v) = x_i$, dem 0-Nachfolgerknoten $low(v)$ und dem 1-Nachfolgerknoten $high(v)$ muß also überprüft werden, ob ein Knoten mit denselben Komponenten schon existiert. Ist dies der Fall, wird kein neuer Knoten erzeugt, sondern der bereits existierende verwendet.
Wir wollen im folgenden die einfache Datenstruktur

```
1  ROBDD ≡
2    begin
3      root : integer;
4      var, low, high : array of integer;
5    end.
```

zur Repräsentation von ROBDDs verwenden. Jeder Knoten eines ROBDDs wird also durch eine natürliche Zahl, seinem Index in der Tabelle, eindeutig bestimmt. Dabei sind 0 und 1 selbstverständlich für die terminalen Knoten reserviert. Der Eintrag *root* bezeichnet den Wurzelknoten des ROBDDs. Die Variablenbeschriftung x_i eines Knotens wird durch ihre Position in der Variablenordnung beschrieben. Zur Vereinfachung der nachfolgenden Algorithmen wollen wir weiterhin vereinbaren, daß die Knoten 0 und 1 den Variablenindex $|\pi| + 1$ zugeordnet bekommen.
Die Abbildung 3.4 verdeutlicht diese Datenstruktur nochmals an einem Beispiel.



u	var	low	$high$
0	4		
1	4		
2	3	0	1
3	3	1	0
4	2	3	2
5	1	2	4

Abbildung 3.4: ROBDD mit entsprechender Datenstruktur.

Die Frage, ob ein Knoten v, repräsentiert durch das Tripel $(i, low[v], high[v])$, schon vorhanden ist, muß durch Suchen in der ROBDD-Datenstruktur beantwortet werden. Um diese Suche effizient zu gestalten, verwendet man eine Hashtabelle $UniqueTable$, die für jeden Hashwert eines Knotentripels einen Verweis auf einen entsprechenden

Knoten im ROBDD liefert. Da es möglich ist, daß verschiedene Tripel auf den gleichen Wert abgebildet werden, implementiert man die $UniqueTable$ als Feld von Kollisionslisten. Wurde eine gute Hashfunktion gewählt, so enthält jede Kollisionsliste nur wenige Elemente, so daß ein zu einem Tripel gehöriger Knoten schnell gefunden werden kann. Wir gehen daher im weiteren bei der Aufwandsabschätzung der Algorithmen von einem konstanten Aufwand $\mathcal{O}(1)$ für die Operationen auf Hashtabellen aus.

Algorithmus 2 (Erzeugung eines ROBDDs aus einem Booleschen Ausdruck)

```
1  BuildROBDD(F, π) ≡
2    B := new(ROBDD);
3    UniqueTable := new(HashTable);
4    max := 1;
5
6    func MakeNode(B, i, v0, v1, max) ≡
7      if v0 = v1 then return v0
8      else
9        tmp := lookup(i, v0, v1);
10       if tmp ≠ −1 then return tmp
11       else
12         max := max + 1;
13         B.var[max] := i;
14         B.low[max] := v0;
15         B.high[max] := v1;
16         insert(i, v0, v1, max);
17         return max
18       fi
19     fi
20   end MakeNode;
21
22   func build(F, i) ≡
23     if i > |π| then
24       if F = false then return 0 else return 1 fi
25     else
26       v0 := build(replaceVarInTerm(F, i, false), i + 1);
27       v1 := build(replaceVarInTerm(F, i, true), i + 1);
28       return MakeNode(B, i, v0, v1, max)
29     fi
30   end build;
31
32 begin
33   B.root := build(F, 1);
34   return B
35 end.
```

Der Algorithmus 2 (BuildROBDD) erzeugt aus einem Booleschen Ausdruck F und einer vorgegebenen Variablenordnung π den zugehörigen ROBDD. Neu zu erzeugende Kno-

ten werden durch die Funktion MakeNode behandelt. Diese Funktion überprüft unter Verwendung der Hashtabelle $UniqueTable$ die Existenz eines Knotens. Ist ein Knoten schon vorhanden, wird dessen Index, ansonsten der Index des neu erzeugten Knotens zurückgegeben. Dadurch wird die Reduziertheit des ROBDDs garantiert. MakeNode verwendet zum Bearbeiten der Hashtabelle $UniqueTable$ die Operationen: lookup und insert. Die Funktion lookup(i, v_0, v_1) sucht nach einem durch das Tripel (i, v_0, v_1) spezifizierten Knoten und gibt dessen Nummer, falls der Knoten existiert, zurück. Existiert ein solcher Knoten nicht, wird -1 zurückgegeben. Die Funktion insert(i, v_0, v_i, u) trägt den noch nicht vorhandenen Knoten u in die Hashtabelle ein.

3.2.4 Operationen auf ROBDDs

Die Grundlage für ein effizientes Arbeiten mit ROBDDs ist eine geschickte Implementierungstechnik der entsprechenden Operatoren.

Binäre Boolesche Operatoren

Sei $\star \in \{ \cdot, + \}$ ein binärer Boolescher Operator und f, g zwei n-stellige Schaltfunktionen in ROBDD-Darstellung. Zur Berechnung der Schaltfunktion $f \star g$ verwendet man die Shannon-Entwicklung bzgl. der in der Variablenordnung führenden Variablen x_i.

$$f \star g = x_i \cdot (f_{x_i} \star g_{x_i}) + \overline{x_i} \cdot (f_{\overline{x_i}} \star g_{\overline{x_i}})$$

Berechnet man nun rekursiv die beiden ROBDDs B_0 und B_1 der Funktionen $f_{\overline{x_i}} \star g_{\overline{x_i}}$ und $f_{x_i} \star g_{x_i}$, so läßt sich die Funktion $f \star g$ einfach dadurch erzeugen, daß man einen Knoten v mit der Beschriftung $var(v) = x_i$ erzeugt und dessen 0-Kante auf B_0 und dessen 1-Kante auf B_1 zeigen läßt.
Achtet man beim Erzeugen neuer Knoten auf die Reduziertheit, d. h. verwendet man die Funktion MakeNode, ist der Ergebnis-ROBDD einer binären Booleschen Operation wiederum ein reduzierter OBDD.
Das hier vorgestellte Verfahren hat den Nachteil, daß durch das rekursive Verfahren zur Erzeugung der Unterbäume alle Shannon-Zerlegungen explizit ausgeführt werden müssen. Man muß also letztendlich 2^n Teilprobleme bearbeiten.
Dies läßt sich mit der folgenden Überlegung effizienter gestalten. Bei jedem rekursiven Aufruf werden zwei Argumente verwendet. Diese Argumente stellen jeweils Unterfunktionen von f und g dar. Jede Unterfunktion wiederum korrespondiert mit genau einem ROBDD-Knoten. Zur Vermeidung von Mehrfachaufrufen mit gleichen Argumentpaaren protokolliert man daher die schon berechneten Resultate in einer Tabelle. Dadurch lassen sich Mehrfachaufrufe durch einfaches Nachsehen in der Tabelle ersetzen. Die ursprünglich exponentielle Anzahl von Zerlegungen läßt sich so durch das Produkt der beiden ROBDD-Größen beschränken.

Satz 13
Seien die beiden Schaltfunktionen f und g durch die ROBDDs F und G bezüglich der gleichen Variablenordnung π repräsentiert. Für jede binäre Boolesche Operation $\star$ kann der ROBDD für die Funktion $f \star g$ bezüglich π in der Zeit $\mathcal{O}(size(F) \cdot size(G))$ bestimmt werden.

Der Algorithmus 3 (BinOP) berechnet die Schaltfunktion $f \star g$ für eine binäre Boolesche Operation $\star$ und zwei als ROBDD repräsentierte Schaltfunktionen f und g. Zur Speicherung der schon berechneten Resultate verwendet der Algorithmus die Tabelle $ResultTable$. Diese läßt sich effizient durch eine Hashtabelle implementieren.

Algorithmus 3 (Binäre Boolesche Operatoren auf ROBDDs)

```
1  BinOP(op, F, G) ≡
2    B := new(ROBDD);
3    UniqueTable := new(HashTable);
4    ResultTable := new(HashTable);
5    max := 1;
6
7    func apply(u1, u2) ≡
8      if ResultTable[u1, u2] ≠ empty then return ResultTable[u1, u2] fi;
9      if u1 ∈ { 0, 1 } ∧ u2 ∈ { 0, 1 } then res := op(u1, u2)
10     elsif F.var[u1] = G.var[u2] then
11       v0 := apply(F.low[u1], G.low[u2]);
12       v1 := apply(F.high[u1], G.high[u2]);
13       res := MakeNode(B, F.var[u1], v0, v1, max)
14     elsif F.var[u1] < G.var[u2] then
15       v0 := apply(F.low[u1], u2);
16       v1 := apply(F.high[u1], u2);
17       res := MakeNode(B, F.var[u1], v0, v1, max)
18     else  /* F.var[u1] > G.var[u2] */
19       v0 := apply(u1, G.low[u2]);
20       v1 := apply(u1, G.high[u2]);
21       res := MakeNode(B, G.var[u2], v0, v1, max)
22     fi
23     ResultTable[u1, u2] := res;
24     return res
25   end apply;
26
27 begin
28   B.root := apply(F.root, G.root);
29   return B
30 end.
```

Komplement

Sei f eine n-stellige Schaltfunktion in ROBDD-Darstellung. Das Komplement $\overline{f}$ der Schaltfunktion ergibt sich durch das Vertauschen der Kanten, die auf die termina-

len Knoten zeigen. Kanten, die auf den 0-Knoten zeigen, werden auf den 1-Knoten umgelenkt und umgekehrt (vgl. Abbildung 3.5).

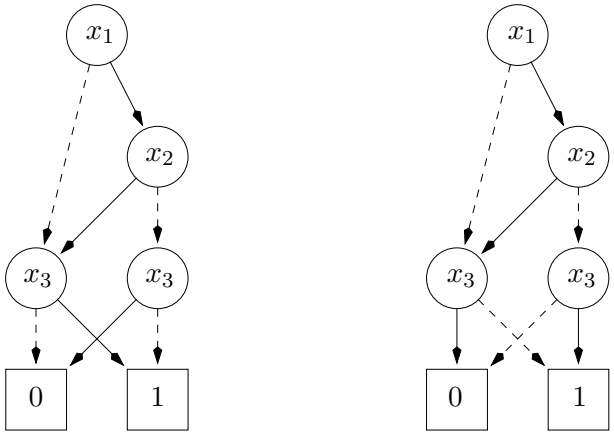


Abbildung 3.5: ROBDDs für die Funktionen f und $\overline{f}$.

Der Algorithmus 4 (Komplement) berechnet das Komplement durch einen Durchlauf des ROBDDs. Dabei wird beim Erreichen eines terminalen Knotens dessen komplementärer Knoten erzeugt und der ROBDD beim Aufsteigen neu zusammengesetzt. Der Aufwand der Komplementbildung ist also durch die ROBDD-Größe beschränkt.

Satz 14
Sei die Schaltfunktion f durch den ROBDD F bezüglich der Variablenordnung π repräsentiert. Der ROBDD des Komplements von f bezüglich der Variablenordnung π kann in der Zeit $\mathcal{O}(size(F))$ bestimmt werden.

Äquivalenztest

Seien f und g zwei n-stellige Schaltfunktionen. Die Äquivalenz der durch die ROBDDs F und G repräsentierten Schaltfunktionen ist durch einen Isomorphietest zu bewerkstelligen. Normalerweise werden dazu beide ROBDDs parallel in einer Tiefensuche von den Wurzeln aus durchlaufen. Bei jedem Übergang zu den 0- bzw. 1-Nachfolgern der beiden aktuellen Knoten wird die Beschriftung der beiden Knoten verglichen. Sind diese für alle Knoten gleich, beschreiben F und G die gleiche Schaltfunktion (vgl. Algorithmus 5 (Äquivalenz)).

Satz 15
Seien die Schaltfunktionen f und g durch die ROBDDs F und G bezüglich der gleichen Variablenordnung π repräsentiert. Der Äquivalenztest für die zwei Darstellungen kann in der Zeit $\mathcal{O}(size(B_1) + size(B_2))$ durchgeführt werden.

Algorithmus 4 (Komplement)

```
1  Komplement(F) ≡
2    B := new(ROBDD);
3    UniqueTable := new(HashTable);
4    ResultTable := new(HashTable);
5    max := 1;
6
7    func compl(u) ≡
8      if u = 0 then return 1 fi;
9      if u = 1 then return 0 fi;
10     if ResultTable[u] ≠ empty then return ResultTable[u] fi;
11     v0 := compl(F.low[u]);
12     v1 := compl(F.high[u]);
13     res := MakeNode(B, F.var[i], v0, v1, max);
14     ResultTable[u] := res;
15     return res
16   end compl;
17
18 begin
19   B.root := compl(F);
20   return B
21 end.
```

Algorithmus 5 (Äquivalenztest)

```
1  Äquivalenz(F, G) ≡
2    ResultTable := new(HashTable);
3
4    func equiv(u1, u2) ≡
5      if u1 ∈ {0, 1} ∨ u2 ∈ {0, 1} then return (u1 = u2) fi;
6      if F.var[u1] ≠ G.var[u2] then return false fi;
7      if ResultTable[u1, u2] ≠ empty then return ResultTable[u1, u2] fi;
8      res := equiv(B.low[u1], B.low[u2]) ∧ equiv(B.high[u1], B.high[u2]);
9      ResultTable[u1, u2] := res;
10     return res
11   end equiv;
12
13 begin
14   return equiv(F.root, G.root)
15 end.
```

Cofaktoren

Die letzte für uns wichtige Operation im Umgang mit ROBDDs ist die Cofaktor-Bildung. Sei f eine n-stellige Schaltfunktion, die durch einen ROBDD F dargestellt ist. Die Berechnung des negativen bzw. positiven Cofaktors bezüglich einer Variablen x_i basiert auf der Suche aller Knoten mit dem Variablenindex i. Wird ein solcher Knoten gefunden, so ersetzt man ihn durch seinen 0- bzw. 1-Nachfolger (vgl. Algorithmus 6 (Cofaktor)). Der sich ergebende ROBDD enthält somit keinen Knoten mit Variablenindex i.

Satz 16
Sei die Schaltfunktion f durch den ROBDD F bezüglich der Variablenordnung π repräsentiert. Der negative bzw. positive Cofaktor von f bezüglich x_i kann in der Zeit $\mathcal{O}(size(F))$ bestimmt werden.

Bottom-Up-Konstrunktion von ROBDDs

Die Verfahren zur Komplementbildung bzw. zur Berechnung von binären Operationen auf ROBDDs bieten eine weitere Möglichkeit, ROBDDs aus einem Booleschen Ausdruck zu konstruieren. Dazu geht man von den elementaren ROBDDs (Abbildung 3.6) aus und setzt diese, basierend auf dem strukturellen Formelaufbau, unter Anwendung der entsprechenden ROBDD-Operationen zusammen.

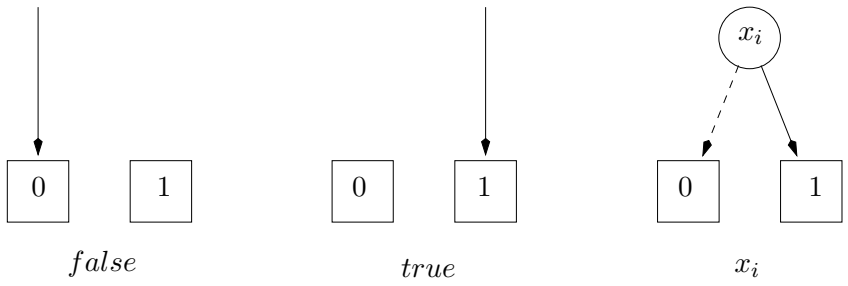


Abbildung 3.6: Elementare ROBDDs

3.2.5 Effiziente Implementierung

Die Geschwindigkeit von Anwendungen, die ROBDDs verwenden, hängt sehr stark von der Effizienz der zugrunde liegenden ROBDD-Datenstruktur und den Operationen auf dieser Datenstruktur ab. Die Arbeit von K. Brace, R. Rudell und R. Bryant [BRB90] bildet die Grundlage für eine Reihe von schnellen und speichereffizienten Implementierungen der grundlegenden ROBDD-Konzepte und -Operationen. Vier der in der Arbeit [BRB90] verwendeten Ansätze wollen wir im folgenden kurz schildern.

Algorithmus 6 (Cofaktor)

```
1  Cofaktor(F, i, value) ≡
2    B := new(ROBDD);
3    UniqueTable := new(HashTable);
4    ResultTable := new(HashTable);
5    max := 1;
6
7    func restrict(u) ≡
8      if u ∈ { 0, 1 } then return u fi;
9      if ResultTable[u] ≠ empty then return ResultTable[u] fi;
10     if (F.var[u] = i) ∧ (value = 0) then res := restrict(F.low[u])
11     elsif (F.var[u] = i) ∧ (value = 1) then res := restrict(F.high[u])
12     elsif F.var[u] < i then
13       v0 := restrict(F.low[u]);
14       v1 := restrict(F.high[u]);
15       res := MakeNode(B, F.var[u], v0, v1, max)
16     else /* F.var[u] > i */
17       v0 := restrict(F.low[u]);
18       v1 := restrict(F.high[u]);
19       res := MakeNode(B, F.var[u] − 1, v0, v1, max)
20     fi
21     ResultTable[u] := res;
22     return res
23   end restrict;
24
25 begin
26   B.root := restrict(F.root);
27   return B
28 end.
```

Shared ROBDDs

Die Idee, gleiche Untergraphen eines ROBDDs nur einmal zu repräsentieren, läßt sich natürlich auch auf mehrere ROBDDs erweitern. Man repräsentiert dazu mehrere Schaltfunktionen in einem einzigen gerichteten azyklischen Graphen mit mehreren Wurzeln. Diese Form der Darstellung wird *Shared ROBDDs* genannt und basiert auf den Arbeiten [BRB90, MIY90].
Die gemeinsame Nutzung von Untergraphen zwischen verschiedenen Funktionen spart Rechenzeit und Speicherplatz.
Man modifiziert dazu die ROBDD-Datenstruktur. Die Komponente *root* wird entfernt und die verbleibenden Tabellen werden von allen ROBDDs gemeinsam genutzt. Jeder ROBDD wird dann nur noch durch einen Index in die Tabellen (auf seinen Wurzelknoten) repräsentiert.
Durch die Nutzung einer gemeinsamen Datenstruktur zur Speicherung aller erzeugten ROBDDs erhalten wir auch einen sehr einfachen und schnellen Äquivalenztest. Da zwei äquivalente Schaltfunktionen die gleiche ROBDD-Darstellung besitzen und gemeinsame Unterfunktionen nur einmal repräsentiert werden, folgt, daß die beiden Schaltfunktionen den gleichen Wurzelknoten besitzen müssen. Ein Äquivalenztest besteht also nur aus einem Vergleich der Wurzelknoten, d. h. aus einem Indexvergleich, was sich in konstanter Zeit ausführen läßt.

Garbage-Collection

Bei unserer getrennten Speicherung der ROBDDs genügt zur Verwaltung der noch freien Knoten eine Variable max, die als Zeiger auf die zuletzt vergebene Position eines Knotens im ROBDD dient.
Bei Operationen auf Shared ROBDDs können Knoten bzw. ganze Unterbäume überflüssig werden. Die von ihnen belegten Knoten befinden sich in der gleichen Datenstruktur wie die noch benötigten Knoten. Daher genügt eine einfache Freispeicherverwaltung nicht mehr.
Man verwendet aus diesem Grund ein *Garbage-Collection-Verfahren*, um den unnötig belegten Speicherplatz zur Wiederverwendung freizugegeben. Das Garbage-Collection-Verfahren markiert zu diesem Zweck alle von den Wurzelknoten erreichbaren Knoten und fügt alle anderen Knoten in einer Liste zusammen. Bei der Erzeugung eines Knotens verwendet man also die Liste der freien Knoten, um seine Position zu ermitteln.

Komplementärkanten

Die ROBDD-Darstellungen einer Schaltfunktion f und deren Komplement $\overline{f}$ unterscheiden sich, wie wir wissen, nur durch die beiden vertauschten terminalen Knoten. Durch die Einführung einer Kantenbeschriftung, die eine Kante als komplementierte Kante auszeichnet, ist es möglich, eine Schaltfunktion und deren Komplement mit dem

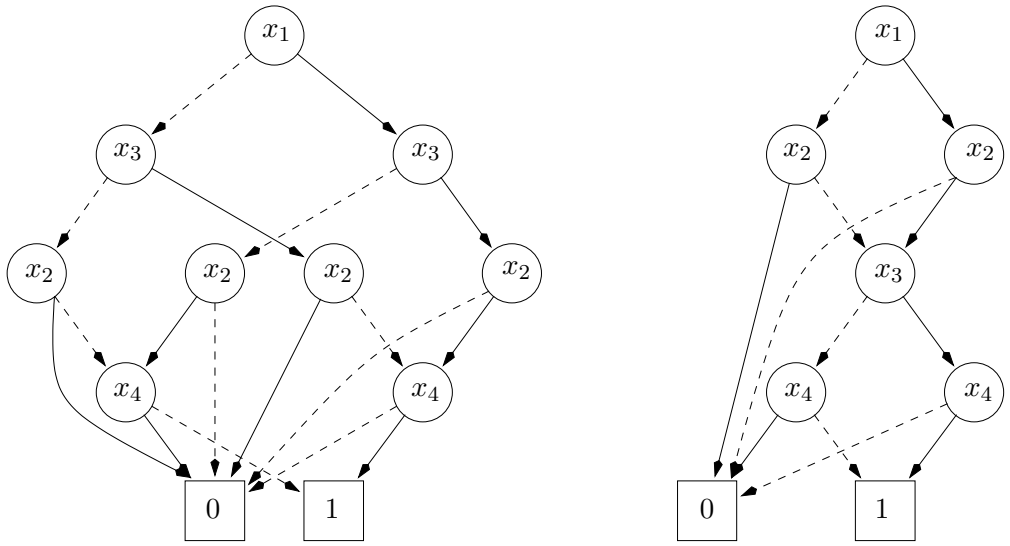
gleichen ROBDD darzustellen. Man interpretiert eine *Komplementärkante* wie folgt: Ist ein Unterbaum eines Knotens über eine normale Kante erreichbar, so wird dieser in der ursprünglichen Weise interpretiert. Ist dagegen ein Unterbaum über eine komplementäre Kante angebunden, so wird er wie das Komplement der durch den Unterbaum repräsentierten Schaltfunktion interpretiert. Das Komplement einer Schaltfunktion f unterscheidet sich also nur durch eine komplementierte Kante auf den Wurzelknoten des f repräsentierenden ROBDDs. Man benötigt daher auch nur einen terminalen Knoten, da sich ein terminaler Knoten durch das Komplement des jeweils anderen darstellen läßt.
Die drei Kanten eines Knotens (eine eingehende und zwei ausgehende Kanten) bieten acht unterschiedliche Möglichkeiten zur Komplementierung an. Es stellt sich heraus, daß jeweils zwei der acht Möglichkeiten äquivalent sind. Aufgrund dieser Redundanz verliert diese Darstellungsvariante die Kanonizität. Schränkt man allerdings die Vergabe der komplementären Kanten ein (z. B. 1-Kanten dürfen nicht komplementiert werden), so erhält man auch mit den komplementierten Kanten eine kanonische Darstellung.
Durch die Einführung von komplementierten Kanten erhält man nicht nur eine Speicherplatzreduzierung, sondern auch eine in konstanter Zeit ausführbare Komplement-Operation.

Cache-basierte Hashtabellen

Die zur Vermeidung von Mehrfachaufrufen mit gleichen Argumenten verwendete Tabelle *ResultTable* wird meist zum Einsparen von Speicherplatz durch eine *cache-basierte Hashtabelle* implementiert. Dadurch ist es nur möglich, eine fest vorgegebene Anzahl von Resultaten zu speichern. Den so eingesparten Speicherplatz erkauft man sich durch einen erhöhten Berechnungsaufwand. Durch das begrenzte Speichern der Resultate besteht die Möglichkeit, daß schon berechnete Ergebnisse vergessen werden und daher später nochmals zu berechnen sind. Im Extremfall kann es dazu führen, daß eine exponentielle Anzahl von Berechnungen nötig ist. Dies tritt aber bei einer geeignet gewählten Hashfunktion (höchst) selten auf.

3.2.6 Variablenordnung

Die Darstellung einer Schaltfunktion mit Hilfe eines ROBDDs setzt eine fest vorgegebene Variablenordnung voraus. Diese kann einen entscheidenden Einfluß auf die Größe der ROBDD-Darstellung und damit auf den Speicher- und Rechenzeitbedarf haben. Die Abbildung 3.7 zeigt zwei äquivalente ROBDDs mit unterschiedlichen Variablenordnungen.
Eine „gute“ Variablenordnung kann zu einer sehr kompakten Darstellung und infolgedessen zu geringen Rechenzeiten führen, während eine ungünstige Variablenordnung



$x_1 < x_3 < x_2 < x_4$ $\qquad$ $x_1 < x_2 < x_3 < x_4$

Abbildung 3.7: Zwei ROBDDs der Funktion $f = (x_1x_2 + \overline{x_1x_2})(x_3x_4 + \overline{x_3x_4})$ mit unterschiedlichen Variablenordnungen.

die Überschreitung des physikalisch vorhandenen Speichers nach sich ziehen und damit einen Abbruch der gesamten Berechnung verursachen kann. Man ist daher bestrebt, möglichst gute Variablenordnungen für ROBDD-Darstellungen zu finden. Im wesentlichen gibt es dafür zwei Vorgehensweisen.
Die erste basiert auf heuristischen Verfahren, die strukturelle Informationen über die Anwendung verwenden, welche als Boolesche Funktion modelliert wird. Sollen zum Beispiel die erreichbaren Zustände eines Petrinetzes als ROBDD repräsentiert werden, so bieten sich Aussagen über die nicht gleichzeitige Belegbarkeit von Stellen [SY96, Spr98] als Heuristik für eine gute Variablenordnung an.
Durch empirische Untersuchungen haben sich folgende Grundregeln für eine günstige Variablenordnung herausgestellt:

1. Variablen, die stark voneinander abhängen, sollten auch in der Variablenordnung nahe beieinander stehen. Man benötigt zur Darstellung der Formel $(x_1 \cdot x_2) + (x_3 \cdot x_4) + \cdots + (x_{2n-1} \cdot x_{2n})$ mit der Variablenordnung $x_1 < x_2 < \cdots < x_{2n-1} < x_{2n}$ nur $2n$ Knoten, wogegen man mit der Variablenordnung $x_1 < x_{n+1} < x_2 < x_{n+2} < \cdots < x_n < x_{2n}$ $(2^{n+1} - 2)$ Knoten benötigt.

2. Variablen, die die Schaltfunktion stark beeinflussen, sollten nahe an den terminalen Knoten liegen, da dort die Knotenanzahl potentiell höher ist.

Das Problem dieses Ansatzes liegt nicht nur in der Schwierigkeit, eine gute Heuristik für eine Problemklasse zu finden, sondern auch darin, daß eine für den endgültigen ROBDD günstige Variablenordnung nicht notwendigerweise optimal für die bei der Konstruktion entstehenden Zwischen-ROBDDs sein muß.
Die zweite Vorgehensweise besteht daher darin, während des Berechnungsprozesses eines ROBDDs die Variablenordnung dynamisch anzupassen. Dazu wird versucht, beim Erreichen eines Speicherlimits durch Permutation der Variablenordnung einen ROBDD mit möglichst geringer Größe zu konstruieren. Diese Variablenordnung wird dann für die nächsten Berechnungsschritte bis zum Wiedererreichen des Speicherlimits verwendet. Leider gibt es kein effizientes Verfahren, welches uns die optimale Umordnung der Variablen angibt, so daß uns nur der Weg des Ausprobierens bleibt. Dies führt dazu, daß die Suche nach einer guten Variablenordnung extrem aufwendig ist. Der Aufwand kann durch Heuristiken, die die Anzahl der Permutationen nach bestimmten Kriterien einschränken, zwar reduziert werden, doch muß man sich dann auch mit suboptimalen Variablenordnungen zufriedengeben. Beispiele für solche Heuristiken findet man in den Arbeiten [FMK91, ISY91, Rud93].
Beide Vorgehensweisen ergänzen sich meist sehr gut, so daß sie häufig kombiniert eingesetzt werden. Man beginnt mit einer guten, aus strukturellen Informationen gewonnenen Variablenordnung und benutzt die dynamische Umordnung, um Korrekturen bzw. Optimierungen für die Zwischenresultate zu erhalten.
Es stellt sich nun die Frage, ob alle Booleschen Funktionen sich durch eine geschickt gewählte Variablenordnung kompakt darstellen lassen, oder ob es Funktionen gibt, für die dies nicht möglich ist. Leider ist es sogar der Fall, daß eine kompakte Darstellungsform für fast alle Booleschen Funktionen nicht möglich ist. Dies belegt der folgende Satz.

Satz 17
Nahezu alle n-stelligen Schaltfunktionen benötigen in ihren ROBDD-Darstellungen mindestens $\frac{2^n}{2n}$ *Knoten. Des weiteren sind nahezu alle Schaltfunktionen nicht sensitiv bezüglich der verwendeten Variablenordnung.*

Beweis: Siehe [LL92]. □

Trotz dieses ernüchternden Ergebnisses erfreuen sich ROBDDs großer Popularität. Dies ist darauf zurückzuführen, daß viele Schaltfunktionen in realen Anwendungen glücklicherweise sehr kompakte Darstellungen erlauben.
Zusammenfassend ergeben sich folgende Faktoren, die die Größe einer als ROBDD repräsentierten Schaltfunktion beeinflussen:

1. Die Art des modellierten Problems.
2. Die Anzahl der benötigten Variablen zur Codierung als ROBDD.
3. Die verwendete Variablenordnung.

3.3 Zusammenfassende Bemerkungen

Binäre Entscheidungsgraphen eignen sich zur Darstellung von Schaltfunktionen. Aber erst durch Hinzunahme von Ordnungsbedingungen und Reduktionsmechanismen erhält man eine kanonische Darstellungsform, die die für die Praxis relevanten Schaltfunktionen kompakt darstellt. Wichtig für die Größe der Darstellung ist meist die Wahl der Variablenordnung. Eine Möglichkeit zur Bestimmung guter Variablenordnungen basiert auf heuristischen Verfahren, die die Struktur des Modells ausnutzen. Durch spezielle Datenstrukturen und Algorithmen ist es möglich, ROBDDs rechnergestützt auf viele Probleme der Informatik anzuwenden. Einer dieser Problemkreise ist die kompakte Darstellung und Handhabung großer Mengen bzw. großer Zustandsräume, z. B. von nebenläufigen Systemen. Diesem Thema widmet sich das nächste Kapitel.

4 Symbolische Petrinetzanalyse

Als Repräsentation der Interleaving-Semantik von Petrinetzen wird der Erreichbarkeitsgraph bei vielen Verfahren zur Verifikation von Petrinetzeigenschaften benutzt. Da die Knoten des Erreichbarkeitsgraphen den Markierungen des Petrinetzes entsprechen, ist man sofort mit dem Problem der Zustandsraumexplosion konfrontiert.
In den letzten Jahren haben sich sogenannte „symbolische“ Verfahren im Umgang mit großen Mengen als sehr erfolgreich erwiesen. Diese Verfahren beschreiben Mengen durch Terme bzw. Formeln und nicht durch ein explizites Aufzählen aller Elemente. Zu diesen Verfahren gehören z. B. die Darstellung von Mengen als Nullstellenmengen von Polynomidealen [Spr97b], so wie auch die im vorherigen Kapitel eingeführte Darstellung als Schaltfunktionen. Diese Art von Beschreibungen hat den Vorteil, daß große Mengen durch meist sehr kleine Umschreibungen (Formeln) bzw. durch Umschreibungen, die sich sehr kompakt darstellen lassen, ausdrückbar sind.
Als eine kompakte und effiziente Darstellung von Schaltfunktionen haben wir schon die ROBDDs kennengelernt. Diese wollen wir im folgenden nutzen, um durch Schaltfunktionen beschriebene Zustandsmengen zu bearbeiten und somit Eigenschaften von Petrinetzen zu verifizieren.
Es sei nochmals darauf hingewiesen, daß wir nur sichere Petrinetze betrachten wollen. Die folgenden Aussagen lassen sich aber auf beliebige beschränkte Petrinetze übertragen.

4.1 Der fundamentale Isomorphismus

Eine Markierung m eines sicheren Petrinetzes $\mathcal{N} = [S, T, F, m_0]$ kann als Menge M von Stellen aus S repräsentiert werden. Hierbei gilt, daß eine Stelle s in M enthalten ist, wenn s bei m markiert ist.
Des weiteren kann jede Menge von Markierungen aus $\mathcal{R}_\mathcal{N}(m_0)$ als Menge $\mathcal{M}$ von Teilmengen von Stellen repräsentiert werden, so daß jede Menge von Stellen $M \in \mathcal{M}$ einer Markierung aus $\mathcal{R}_\mathcal{N}$ entspricht. Bezeichne 2^S die Potenzmenge von S. Dann ist das Tupel $(2^{(2^S)}, \cup, \cap, ^-, \emptyset, 2^S)$ die endliche Boolesche Algebra der Mengen von (sicheren) Markierungen von $\mathcal{N}$. Diese Mengenalgebra ist isomorph zu der durch die n-stellige Schaltfunktionen induzierten Booleschen Algebra (vgl. Satz 9).
Wir können daher s sowohl als Stelle aus S sowie auch als Variable in der Booleschen

Algebra der n-stelligen Schaltfunktionen betrachten. Ebenso bezeichnet M sowohl eine Markierung sowie auch die Menge der bei einer Markierung markierten Stellen.
Jede Markierung $M \in 2^S$ läßt sich also eindeutig durch einen Vektor aus $\mathbb{B}^n$ darstellen. Die bijektive Abbildung $\mathcal{C}\colon 2^S \longrightarrow \mathbb{B}^n$, $M \mapsto (s_1, \ldots, s_n)$ mit

$$s_i := \begin{cases} 0, & \text{falls } s_i \notin M \\ 1, & \text{falls } s_i \in M \end{cases}$$

verdeutlicht diese Beziehung. Betrachtet man z. B. die Initialmarkierung $\{ s_1, s_6 \}$ des Petrinetzes aus Abbildung 2.1, so läßt sich diese sowohl als Vektor $(1, 0, 0, 0, 0, 1) \in \mathbb{B}^6$ sowie auch als Boolescher Ausdruck $s_1 \wedge \neg s_2 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge s_6$ darstellen.
Durch die Bijektion $\mathcal{C}$ kann man jeder Menge von Markierungen $\mathcal{M} \subseteq 2^S$ die korrespondierende Menge von Vektoren $\mathcal{V} \subseteq \mathbb{B}^n$ durch

$$\mathcal{V} := \{ v \in \mathbb{B}^n \mid \exists M \in \mathcal{M},\ v = \mathcal{C}(M) \}$$

zuordnen. Dadurch können wir die charakteristische Funktion $\chi_{\mathcal{M}}$ von $\mathcal{M}$ als eine n-stellige Schaltfunktion definieren, die als Resultat für alle Vektoren, die einer Markierung aus $\mathcal{M}$ entsprechen, eine 1 ergibt. Es gilt also $\chi_{\mathcal{M}} = \chi_{\mathcal{V}}$.
Alle Mengenoperationen auf Markierungsmengen können also als logische Operationen ihrer charakteristischen Funktionen betrachtet werden. Seien $\mathcal{M}, \mathcal{M}' \subseteq 2^S$ dann gilt:

- $\chi_{\mathcal{M} \cup \mathcal{M}'} = \chi_{\mathcal{M}} \vee \chi_{\mathcal{M}'}$
- $\chi_{\mathcal{M} \cap \mathcal{M}'} = \chi_{\mathcal{M}} \wedge \chi_{\mathcal{M}'}$
- $\chi_{\overline{\mathcal{M}}} = \overline{\chi_{\mathcal{M}}} \wedge \chi_{2^S}$

Dies ermöglicht uns Markierungsmengen durch ihre charakteristischen Funktionen als ROBDD darzustellen. Wir erhalten dadurch eine kompakte und effiziente Darstellungsform für Mengen von Markierungen, die uns gleichzeitig schnelle Operationen zur Verarbeitung bietet.
Dieser Ansatz läßt sich auch zur Repräsentation binärer Relationen zwischen Markierungsmengen verwenden. Eine solche binäre Relation $R \subseteq \mathcal{M} \times \mathcal{M}'$ benötigt zur Darstellung der beiden unterschiedlichen Markierungsmengen verschiedene Variablen. Seien daher $\{ s_1, \ldots, s_n \}$ die Booleschen Variablen von $\mathcal{M}$ und $\{ s'_1, \ldots, s'_n \}$ die Booleschen Variablen von $\mathcal{M}'$. Die charakteristische Funktion der Relation R ist definiert als

$$\chi_R(s_1, \ldots, s_n, s'_1, \ldots, s'_n) = 1 \Longleftrightarrow \\ \exists (M, M') \in R\ [\mathcal{C}(M) = (s_1, \ldots, s_n) \wedge \mathcal{C}(M') = (s'_1, \ldots, s'_n)].$$

Die Menge der Markierungen aus $\mathcal{M}$, die in Relation zu Markierungen aus $\mathcal{M}'$ stehen, wird wie folgt definiert:

$$R(\mathcal{M}) := \{\, M \in \mathcal{M} \mid \exists M' \in \mathcal{M}',\ (M, M') \in R \,\}.$$

Die charakteristische Funktion $\chi_{R(\mathcal{M})}$ von $R(\mathcal{M})$ ergibt sich durch:

$$\chi_{R(\mathcal{M})}(s_1, \ldots, s_n) := \exists_{s'_1,\ldots,s'_n} \chi_R(s_1, \ldots, s_n, s'_1, \ldots, s'_n).$$

Analog läßt sich auch die Menge der Markierungen aus $\mathcal{M}'$, die in Relation zu Markierungen aus $\mathcal{M}$ stehen, definieren. Ihre charakteristische Funktion $\chi_{R(\mathcal{M}')}$ ergibt sich dann durch:

$$\chi_{R(\mathcal{M}')}(s'_1, \ldots, s'_n) := \exists_{s_1,\ldots,s_n} \chi_R(s_1, \ldots, s_n, s'_1, \ldots, s'_n).$$

4.2 Symbolische Manipulation von Petrinetzen

Die Struktur eines Petrinetzes definiert eine Menge von *Zustandsübergangsfunktionen*, die wiederum das dynamische Verhalten des Petrinetzes bestimmen. Die Zustandsübergangsfunktion einer Transition $t \in T$ ist eine Funktion der Form:

$$\delta\colon 2^S \times T \longrightarrow 2^S, \quad \delta(M,t) := \begin{cases} M', & \text{falls } M \xrightarrow{t} M' \\ \emptyset, & \text{sonst} \end{cases}$$

welche eine Markierung $M \in 2^S$ in eine Markierung $M' \in 2^S$ durch Schalten von $t \in T$ überführt. Dabei nehmen wir an, daß t bei M schaltfähig ist, ansonsten erhält man als Ergebnis $\delta(M,t) = \emptyset$. Dies entspricht der Ein-Schritt-Erreichbarkeit bei Petrinetzen. Das Schalten von Transitionen läßt sich von einzelnen Markierungen auch auf Markierungsmengen erweitern.

$$\mathit{post}\colon 2^{(2^S)} \times T \longrightarrow 2^{(2^S)}, \quad \mathit{post}(\mathcal{M},t) := \{\, M' \in 2^S \mid \exists M \in \mathcal{M},\ M \xrightarrow{t} M' \,\}$$

Die Funktion *post* überführt eine Markierungsmenge $\mathcal{M}$ in eine Menge von Markierungen $\mathcal{M}'$, die aus $\mathcal{M}$ durch Schalten von t erreichbar sind. Wir verwenden dabei häufig die Schreibweise: $\mathit{post}_t(\mathcal{M}) := \mathit{post}(\mathcal{M},t)$. Das Schaltverhalten des gesamten Petrinetzes ergibt sich dann als die Funktion:

$$\mathit{post}_T\colon 2^{(2^S)} \longrightarrow 2^{(2^S)}, \quad \mathit{post}_T(\mathcal{M}) := \bigcup_{t \in T} \mathit{post}_t(\mathcal{M}).$$

Diese vorwärts gerichtete Beschreibung der Petrinetzdynamik hat ein rückwärts gerichtetes Komplement, welches durch die Funktionen

$$pre\colon 2^{(2^S)} \times T \longrightarrow 2^{(2^S)}, \quad pre(\mathcal{M}, t) := \{\, M' \in 2^S \mid \exists M \in \mathcal{M},\ M' \xrightarrow{t} M \,\}$$

$$pre_T\colon 2^{(2^S)} \longrightarrow 2^{(2^S)}, \qquad pre_T(\mathcal{M}) := \bigcup_{t \in T} pre_t(\mathcal{M})$$

beschrieben wird. Dabei beschreibt $pre_t(\mathcal{M})$ (bzw. $pre_T(\mathcal{M})$) die Menge der Markierungen, die in einem Schritt durch Schalten von t (bzw. einer beliebigen Transition aus T) eine Markierung aus $\mathcal{M}$ erreichen.
Markierungsmengen lassen sich, wie schon beschrieben, durch ihre charakteristische Funktion repräsentieren und sind daher durch die Standardoperationen der Booleschen Algebra manipulierbar.
Im wesentlichen gibt es zwei verschiedenen Arten, wie sich die Zustandsübergangsfunktionen und damit die Dynamik eines Petrinetzes mit Booleschen Operationen modellieren lassen, die in den beiden folgenden Abschnitten beschrieben werden.

4.2.1 Zustandsübergangsrelationen

Der erste Ansatz beschreibt die Zustandsübergangsfunktionen als Zustandsübergangsrelationen (vgl. [PRCB94]). Eine Zustandsübergangsrelation ist dabei eine binäre Relation, die den Bezug zwischen Vor- und Nachzuständen beschreibt. Dazu benötigen wir zwei unterschiedliche Sätze von Booleschen Variablen zur Beschreibung der Stellenbelegungen. Wir wollen $\{ s_1, \ldots, s_n \}$ für die Bezeichnung der Booleschen Variablen der Vorzustände und $\{ s'_1, \ldots, s'_n \}$ für die Nachzustände verwenden. Die Zustandsübergangsrelation für eine Transition $t \in T$ läßt sich dann wie folgt definieren:

$$TR_t(s_1, \ldots, s_n, s'_1, \ldots, s'_n) := \bigwedge_{s_i \in {}^\bullet t} s_i \wedge \bigwedge_{s_i \in t^\bullet} s'_i \wedge \bigwedge_{s_i \in {}^\bullet t - t^\bullet} \neg s'_i \wedge \bigwedge_{s_i \notin {}^\bullet t \cup t^\bullet} (s_i = s'_i).$$

Der erste Faktor hat die Bedeutung, daß alle Vorstellen von t vor dem Schalten markiert sein müssen. Der zweite und dritte Faktor beschreiben die Markierungen der Stellen nach dem Schalten, wobei alle Nachstellen markiert sein müssen und alle echten Vorstellen keine Marke mehr enthalten dürfen. Der letzte Faktor besagt, daß alle anderen Stellen unverändert bleiben. Wir verwenden dafür die abkürzende Schreibweise $a = b :\Longleftrightarrow (a \wedge b) \vee (\neg a \wedge \neg b)$. Da wir nur sichere Petrinetze betrachten, erübrigt sich die Forderung nach unmarkierten Nachstellen einer schaltfähigen Transition.
Aus den Zustandsübergangsrelationen ergeben sich die Zustandsübergangsfunktionen bezüglich der charakteristischen Funktion der Markierungsmenge $\mathcal{M}$ wie folgt:

$$
\begin{aligned}
pre_t(\chi_{\mathcal{M}}) &:= \exists_{s'_1,\ldots,s'_n}(TR_t \wedge \chi_{\mathcal{M}}[s_i \leftarrow s'_i]) \\
pre_T(\chi_{\mathcal{M}}) &:= \exists_{s'_1,\ldots,s'_n}(\bigvee_{t\in T} TR_t \wedge \chi_{\mathcal{M}}[s_i \leftarrow s'_i]) \\
post_t(\chi_{\mathcal{M}}) &:= \exists_{s_1,\ldots,s_n}(TR_t \wedge \chi_{\mathcal{M}})[s'_i \leftarrow s_i] \\
post_T(\chi_{\mathcal{M}}) &:= \exists_{s_1,\ldots,s_n}(\bigvee_{t\in T} TR_t \wedge \chi_{\mathcal{M}})[s'_i \leftarrow s_i]
\end{aligned}
$$

Die Definition der Zustandsübergangsfunktionen benötigt Variablenumbenennungen. Diese werden durch $[x \leftarrow y]$ ausgedrückt, was bedeuten soll, daß die Variable x durch die Variable y ersetzt wird. Eine solche Umbenennung einer Variablen in einer Schaltfunktion f läßt sich durch Boolesche Operationen als $f[x \leftarrow y] := (y \cdot f_x) + (\overline{y} \cdot f_{\overline{x}})$ definieren.

Beispiel 3 (Schalten von t_2, basierend auf der Transitionsübergangsrelation TR_{t_2})
Wir betrachten das Petrinetz aus Abbildung 2.1 mit den Markierungen $\mathcal{M} = \{\{s_1, s_6\}, \{s_2, s_3, s_6\}, \{s_2, s_5, s_6\}\}$ und deren charakteristischen Funktion

$$
\begin{aligned}
\chi_{\mathcal{M}} = \; & (s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge s_6) \quad \vee \\
& (\neg s_1 \wedge s_2 \wedge s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge s_6) \quad \vee \\
& (\neg s_1 \wedge s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge s_5 \wedge s_6).
\end{aligned}
$$

Die Transitionsübergangsrelation der zu schaltenden Transition t_2 ist

$$
\begin{aligned}
& TR_{t_2}(s_1, \ldots, s_n, s'_1, \ldots, s'_n) := \\
& \quad s_2 \wedge s_6 \wedge s'_4 \wedge s'_6 \wedge \neg s'_2 \wedge (s_1 = s'_1) \wedge (s_3 = s'_3) \wedge (s_5 = s'_5).
\end{aligned}
$$

Daraus ergibt sich die charakteristische Funktion $\chi_{\mathcal{M}''}$ der Menge der Folgemarkierungen $\mathcal{M}''$ von $\mathcal{M}$ wie folgt:

$$
\begin{aligned}
\chi_{\mathcal{M}''} &= post_{t_2}(\chi_{\mathcal{M}}) \\
&= \exists s_1 \exists s_2 \exists s_3 \exists s_4 \exists s_5 \exists s_6 (TR_{t_2}(s_1, \ldots, s_n, s'_1, \ldots, s'_n) \wedge \chi_{\mathcal{M}}) \\
&\quad [s'_1 \leftarrow s_1][s'_2 \leftarrow s_2][s'_3 \leftarrow s_3][s'_4 \leftarrow s_4][s'_5 \leftarrow s_5][s'_6 \leftarrow s_6] \\
&= (\neg s_1 \wedge \neg s_2 \wedge s_3 \wedge s_4 \wedge \neg s_5 \wedge s_6) \vee (\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge s_4 \wedge s_5 \wedge s_6).
\end{aligned}
$$

Als Ergebnis erhalten wir also die Folgemarkierungen $\mathcal{M}'' = \{\{s_3, s_4, s_6\}, \{s_4, s_6\}\}$.

4.2.2 Direkte Beschreibung der Zustandsübergangsfunktionen

Der zweite Ansatz beschreibt Zustandsübergangsfunktionen direkt durch Boolesche Operationen [PRCB94]. Man geht auch hier von einer durch ihre charakteristische Funktion beschriebene Markierungsmenge aus und wendet darauf die Booleschen Operationen Disjunktion, Konjunktion, Komplement und Cofaktorbildung an.

$$
\begin{aligned}
pre_t(\chi_{\mathcal{M}}) &:= \left[\chi_{\mathcal{M}\{\,s_i|s_i\in t\bullet\,\}} \wedge \bigwedge_{s_i\in t\bullet} \neg s_i\right]_{\{\,\overline{s_i}|s_i\in\bullet t\,\}} \wedge \bigwedge_{s_i\in\bullet t} s_i \\
pre_T(\chi_{\mathcal{M}}) &:= \bigvee_{t\in T} pre_t(\chi_{\mathcal{M}}) \\
post_t(\chi_{\mathcal{M}}) &:= \left[\chi_{\mathcal{M}\{\,s_i|s_i\in\bullet t\,\}} \wedge \bigwedge_{s_i\in\bullet t} \neg s_i\right]_{\{\,\overline{s_i}|s_i\in t\bullet\,\}} \wedge \bigwedge_{s_i\in t\bullet} s_i \\
post_T(\chi_{\mathcal{M}}) &:= \bigvee_{t\in T} post_t(\chi_{\mathcal{M}})
\end{aligned}
$$

Die hier verwendete abkürzende Schreibweise $\chi_{\mathcal{M}\{\,s_1,s_2,\ldots\,\}}$ bzw. $\chi_{\mathcal{M}\{\,\overline{s_1},\overline{s_2},\ldots\,\}}$ beschreibt die Nacheinanderausführung der positiven bzw. negativen Cofaktorbildung der Schaltfunktion $\chi_{\mathcal{M}}$ bezüglich der Variablen $\{s_1, s_2, \ldots\}$.
Wir wollen anhand eines Beispiels erklären, wie die Zustandsübergangsfunktion $post_t$ das Schalten einer Transition t „simuliert“.

Beispiel 4 (Simulation des Schaltens von t_2 durch $post_{t_2}$)
Wir betrachten das Petrinetz aus Abbildung 2.1 mit den Markierungen $\mathcal{M} = \{\,\{s_1, s_6\}, \{s_2, s_3, s_6\}, \{s_2, s_5, s_6\}\,\}$ und deren charakteristischen Funktion

$$
\begin{aligned}
\chi_{\mathcal{M}} = {} & (s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge s_6) \quad \vee \\
& (\neg s_1 \wedge s_2 \wedge s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge s_6) \quad \vee \\
& (\neg s_1 \wedge s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge s_5 \wedge s_6).
\end{aligned}
$$

Die zu schaltende Transition sei t_2. Im ersten Schritt berechnen wir den Cofaktor von $\chi_{\mathcal{M}}$ bezüglich der Vorstellen s_2 und s_6 von t_2.

$$
\chi_{\mathcal{M}\{s_2,s_6\}} = (\neg s_1 \wedge s_3 \wedge \neg s_4 \wedge \neg s_5) \vee (\neg s_1 \wedge \neg s_3 \wedge \neg s_4 \wedge s_5)
$$

Wir erhalten dadurch alle Markierungen von $\mathcal{M}$, bei denen die Vorstellen s_2 und s_6 markiert sind, und entfernen anschließend aus der Beschreibung dieser Menge die Variable s_2 und s_6. Im nächsten Schritt schränken wir uns auf die Menge der Markierungen ein, bei denen die Vorstellen nicht mehr markiert sind.

$$
\begin{aligned}
\chi_{\mathcal{M}\{s_2,s_6\}} \wedge \neg s_2 \wedge \neg s_6 = {} & (\neg s_1 \wedge \neg s_2 \wedge s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge \neg s_6) \quad \vee \\
& (\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4 \wedge s_5 \wedge \neg s_6)
\end{aligned}
$$

Für die Nachstellen s_4 und s_6 der Transition t_2 verfahren wir analog wie für die Vorstellen. Wir suchen alle Markierungen, bei denen die Nachstellen nicht markiert sind, und entfernen anschließend aus der Beschreibung dieser Menge die entsprechenden negierten Variablen.

$$[\chi_{\mathcal{M}\{s_2,s_6\}} \wedge \neg s_2 \wedge \neg s_6]_{\{\overline{s_4},\overline{s_6}\}} = (\neg s_1 \wedge \neg s_2 \wedge s_3 \wedge \neg s_5) \vee (\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge s_5)$$

Zuletzt schränken wir uns auf die Menge von Markierungen ein, bei denen die Nachstellen markiert sind.

$$\begin{aligned}[\chi_{\mathcal{M}\{s_2,s_6\}} \wedge \neg s_2 \wedge \neg s_6]_{\{\overline{s_4},\overline{s_6}\}} \wedge s_4 \wedge s_6 \quad = \quad & (\neg s_1 \wedge \neg s_2 \wedge s_3 \wedge s_4 \wedge \neg s_5 \wedge s_6) \quad \vee \\ & (\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge s_4 \wedge s_5 \wedge s_6)\end{aligned}$$

Als Ergebnis erhalten wir, wie erwartet, die Folgemarkierungen $\{\,\{s_3, s_4, s_6\}, \{s_4, s_6\}\,\}$.

Der erste Ansatz verwendet Relationen und hat damit den großen Nachteil, daß er doppelt soviele Variablen benötigt wie der zweite direkte Ansatz. Dies führt meist zu einer größeren ROBDD-Darstellung, was sich auf die Effizienz negativ auswirkt. Aus diesem Grund wollen wir im weiteren Teil dieser Arbeit nach dem zweiten Ansatz vorgehen und diesen noch weiter verbessern.

4.2.3 Menge der erreichbaren Markierungen

Mit den oben definierten Zustandsübergangsfunktionen läßt sich nun sehr einfach die Menge der von M_0 aus erreichbaren Markierungen eines Petrinetzes $\mathcal{N}$ berechnen. Dies verdeutlicht der Algorithmus 7 (ReachableSet).

Algorithmus 7 (Zustandsmenge)

```
1  ReachableSet([S, T, F, M_0]) ≡
2
3    func FwdReach(M) ≡
4      Old := M;
5      repeat
6        New := Old;
7        forall t ∈ T do                    /* New := Old ∪ post_T(Old) */
8          New := New ∪ post_t(Old)
9        od
10     until (Old = New);
11     return New
12   end FwdReach;
13
14 begin
15   return FwdReach({ M_0 })
16 end.
```

Zum leichteren Verständnis werden Mengen und Mengenoperationen verwendet. In einer Implementierung werden natürlich die Markierungsmengen über ihre charakteristische Funktion als ROBDD repräsentiert und die Mengenoperationen durch die entsprechenden ROBDD-Operationen ersetzt.

Das bei der Konstruktion der Zustandsmenge verwendete Verfahren basiert auf einem *symbolischen Breitentraversierungsverfahren*. In jedem Schritt werden dabei alle Markierungen berechnet, die von einer bestehenden Markierungsmenge aus durch das Schalten einer Transition des Petrinetzes erreichbar sind.

Wir wollen die dazu maßgebliche Funktion FwdReach etwas näher betrachten.

Die Funktion FwdReach berechnet alle von einer Markierungsmenge $\mathcal{M}$ aus erreichbaren Markierungen. Sie basiert dabei auf einer *Fixpunktiteration*, die in jedem Schritt, ausgehend von den schon berechneten Markierungen, die neu erreichbaren Markierungen berechnet. Wird keine neue Markierung (d. h. keine, die nicht bereits in einem vorherigen Schritt erzeugt wurde) mehr generiert, haben wir den Fixpunkt der Berechnung erreicht und das Verfahren terminiert.

Im i-ten Iterationsschritt werden zuerst alle von Old_i durch das Schalten einer beliebigen Transition erreichbaren Markierungen erzeugt. Diese werden zusammen mit den schon in einem früheren Schritt berechneten Markierungen in New_i akkumuliert. Die Iteration endet, wenn in einem Iterationsschritt keine neue Markierung mehr erzeugt wird.

Diese Verfahren nennt man auch *Vorwärtsiterationsverfahren*, da bei jeder Iteration alle von einer Markierungsmenge in einem Schritt erreichbaren Markierungen berechnet werden. Das Vorwärtsiterationsverfahren ist daher mit den Zustandsübergangsfunktionen $post_t$ (bzw. $post_T$) verbunden. Ersetzt man die Zustandsübergangsfunktionen $post_t$ (bzw. $post_T$) durch die Zustandsübergangsfunktionen pre_t (bzw. pre_T), erhält man ein *Rückwärtsiterationsverfahren*. Die so erzeugte Funktion BwdReach berechnet zu einer Markierungsmenge $\mathcal{M}$ alle Markierungen, die durch das Schalten einer Transitionsfolge eine Markierung aus $\mathcal{M}$ erreichen.

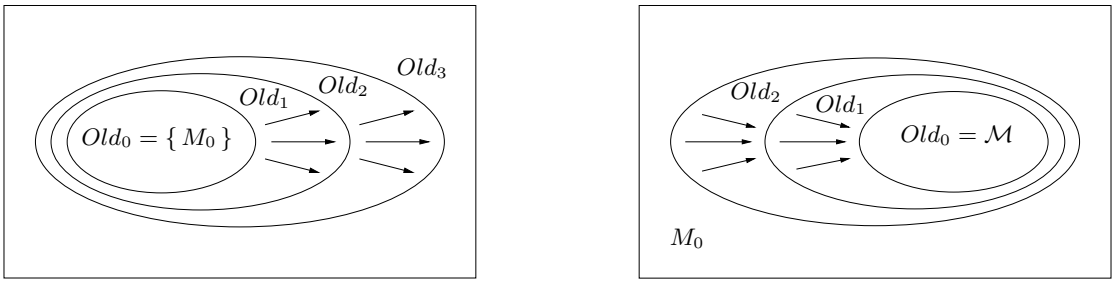


Abbildung 4.1: Vorwärts- und Rückwärtsiteration.

4.2.4 Petrinetzeigenschaften

Eine wichtige Gruppe von Systemeigenschaften, die sogenannten „Sicherheitseigenschaften“, lassen sich auf die Erreichbarkeit bzw. Nichterreichbarkeit von bestimmten Zuständen zurückführen. Der Begriff „Sicherheitseigenschaften“ bezieht sich darauf, daß ein System als sicher angesehen wird, wenn bestimmte ungewollte Zustände, wie z. B. Zustände, bei denen wichtige Systemkomponenten ausgefallen sind, nicht auftreten können. Überträgt man dies auf das Petrinetzmodell, so möchte man nachweisen, daß von M_0 aus Markierungen einer bestimmten Markierungsmenge $\mathcal{M}$ nicht erreichbar sind. Dies läßt sich unter Verwendung der Funktion $\mathsf{FwdReach}$ wie folgt beschreiben und damit überprüfen.

$$\mathcal{M} \cap \mathsf{FwdReach}(\{M_0\}) = \emptyset$$

Wir wollen diesen Ansatz am Problem der Nichterreichbarkeit von toten Markierungen verdeutlichen.

Beispiel 5 (Tote Markierungen)
Im ersten Schritt müssen wir dazu die Menge $\mathcal{D}$ aller toten Markierungen des Petrinetzes $\mathcal{N}$ beschreiben. Eine Transition t ist bei einer Markierung M nicht schaltfähig, wenn mindestens eine Vorstelle bei M nicht markiert ist.

$$\mathcal{D}_t := \bigvee_{s_i \in {}^\bullet t} \neg s_i$$

Eine Markierung ist tot, wenn man bei ihr keine Transition schalten kann.

$$\mathcal{D}_T := \bigwedge_{t \in T} \mathcal{D}_t$$

Die Nichterreichbarkeit von toten Zuständen läßt sich somit durch die folgende Formel ausdrücken.

$$\mathcal{D}_T \cap \mathsf{FwdReach}(\{M_0\}) = \emptyset$$

Das Nichterreichbarkeitsproblem für Petrinetze läßt sich auch unter Verwendung der Rückwärtsiteration ausdrücken. Dazu werden erst alle Markierungen berechnet, die eine Markierung aus $\mathcal{M}$ erreichen können. Danach wird überprüft, ob die Initialmarkierung M_0 zu dieser Menge von berechneten Markierungen gehört.

$$\{M_0\} \cap \mathsf{BwdReach}(\mathcal{M}) = \emptyset$$

Für die Nichterreichbarkeit von toten Zuständen ergibt sich also

$$\{M_0\} \cap \mathsf{BwdReach}(\mathcal{D}_T) = \emptyset.$$

Andere Petrinetzeigenschaften, wie z. B. die Reversibilität, lassen sich als Kombination von Vor- und Rückwärtsiterationen ausdrücken.

Beispiel 6 (reversibel)
Ein Petrinetz $\mathcal{N}$ wird als reversibel bezeichnet, wenn von jeder erreichbaren Markierung aus wieder die Initialmarkierung M_0 erreichbar ist. Man muß dazu die folgende Formel überprüfen.

$$\mathsf{FwdReach}(\{M_0\}) \cap \mathsf{BwdReach}(\{M_0\}) = \mathsf{FwdReach}(\{M_0\})$$

4.2.5 Verbesserungen

Die symbolische Petrinetzanalyse ermöglicht es, Eigenschaften von Petrinetzen mit Hilfe von binären Entscheidungsgraphen zu überprüfen. Die Effizienz dieses Ansatzes hängt sehr stark von der Größe der entstehenden binären Entscheidungsgraphen ab. Im folgenden wollen wir einige Ansätze besprechen, die versuchen, unter Verwendung von zusätzlichen Informationen (z. B. spezielle Struktureigenschaften der zu analysierenden Petrinetze) die BDD-Größe zu beeinflussen.

Transitionsverkettung

Der Erreichbarkeitsalgorithmus 7 (ReachableSet) wendet die Transitionen einzeln auf die gleiche Ausgangszustandsmenge *Old* an. Dies entspricht einer Breitentraversierung des Zustandsraumes. Diese Vorgehensweise benötigt z. B. für eine sequentielle Markierungsfolge von n unterschiedlichen Markierungen n Durchläufe durch die Menge der Transitionen. Dabei werden bei jedem Durchlauf unnötigerweise $|T| - 1$ Transitionen betrachtet. Ordnet man die Schaltreihenfolge der Transitionen bei einem Transitionsdurchlauf entsprechend der zu erzeugenden Markierungsfolge und stellt dabei jeder Transition die durch die vorherige Transition erzeugte Markierung zur Verfügung, reichen meist deutlich weniger Transitionsdurchläufe, um die Markierungsfolge zu erzeugen.
Diese Überlegungen nutzten E. Pastor u. a. [PRCB94] zu einer leichten Veränderung des Erreichbarkeitsalgorithmus 7 (ReachableSet).

```
7  forall  t ∈ T  do                    7  forall  t ∈ T  do
8    New := New ∪ post_t(Old)    ⟹     8    New := New ∪ post_t(New)
9  od                                   9  od
```

Beispiel 7
Für die Markierungsfolge

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_3} M_2 \xrightarrow{t_5} M_3 \xrightarrow{t_2} M_4 \xrightarrow{t_3} M_5 \xrightarrow{t_6} M_6 \xrightarrow{t_1} M_7 \xrightarrow{t_4} M_8$$

benötigt man im alten Verfahren die acht Transitionsdurchläufe $(t_1)(t_3)(t_5)(t_2)(t_3)(t_6)$ $(t_1)(t_4)$, wogegen man im neuen Verfahren mit der Transitionsabfolge $t_1, t_2, t_3, t_4, t_5, t_6$ nur die drei Durchläufe $(t_1t_3t_5)(t_2t_3t_6)(t_1t_4)$ benötigt.

Der wichtigste Punkt dieses Verfahrens ist das Erstellen einer guten Transitionsabfolge zur Erzeugung der Zustandsmenge $\mathcal{R}_{\mathcal{N}}(M_0)$. Dazu kann man die Beobachtung nutzen, daß alle Transitionen, die in der Vortransitionsmenge $\bullet(\bullet t)$ einer Transition t liegen, vor dieser schalten müssen.
Das einfache heuristische Verfahren von A. Noack [Noa99] basiert auf dieser Beobachtung. Für dieses Verfahren werden zunächst alle unter der Initialmarkierung schaltfähigen Transitionen ausgewählt. Im nächsten Schritt markiert man alle Nachstellen dieser Transitionen (die Vorstellen bleiben dabei markiert) und wählt wieder die jetzt schaltfähigen Transitionen aus. Dies führt man fort, bis keine Transitionen mehr schalten können. Bleiben Transitionen übrig, so sind dies Transitionen, die unter keiner erreichbaren Markierung schalten können. Sie müssen daher beim Erzeugen von Zuständen auch nicht berücksichtigt werden.
Der Vorteil des Transitionsverkettungsverfahrens liegt nicht allein in einer Reduktion der Anzahl der Transitionsdurchläufe, sondern auch, wie die Arbeiten von E. Pastor u. a. [PRCB94] und A. Noack [Noa99] zeigen, in einer Reduktion der maximalen ROBDD-Größe und damit des gesamten Speicherplatzbedarfs.
Im Vergleich zu der klassischen Breitentraversierung ergibt sich daraus eine um mehrere Größenordnungen reduzierte Bearbeitungszeit.

Spezielle ROBDD-Operationen

Das Schalten von Transitionen in Petrinetzen läßt sich durch Boolesche Funktionen und ihre ROBDD-Operationen modellieren. Dabei wird das Schalten einer Transition in mehrere ROBDD-Operationen unterteilt, die jeweils Zwischenergebnisse für die nächste Operation erzeugen. Diese Zwischenergebnisse (ROBDDs) enthalten oft viele Knoten, die im Endergebnis nicht mehr benötigt werden. Aus Effizienzgründen liegt es daher nahe, das Schalten einer Transition t direkt als eigene ROBDD-Operation (Algorithmus 8 (Fire)) zu implementieren und somit die Erzeugung unnötiger Knoten in den Zwischenergebnissen zu verhindern.
Man kann diese Idee noch erweitern, indem man zusätzlich zum Schalten einer Transition die Vereinigung mit einer vorgegebenen Menge integriert. Dadurch läßt sich der komplette Iterationsschritt des Schaltens einer Transition durch eine ROBDD-Operation (FireUnion) ohne die Erzeugung von Zwischenergebnissen ausdrücken. Der Algorithmus 7 (ReachableSet) läßt sich also wie folgt umschreiben.

```
7  forall  t ∈ T  do                    7  forall  t ∈ T  do
8    New := New ∪ post_t(Old)    ⟹     8    New := FireUnion(t, Old, New)
9  od                                   9  od
```

Algorithmus 8 (Direktes Schalten einer Transition)

```
1  Fire(F, [S, T, F, M_0], t) ≡
2    B := new(ROBDD);
3    UniqueTable := new(HashTable);
4    ResultTable := new(HashTable);
5    max := 1;
6
7    func fire(u, svec) ≡
8      if svec = () ∨ u = 0 then return u fi;
9      if ResultTable[u] ≠ empty then return ResultTable[u] fi;
10     s := head(svec);
11     if F.var[u] < s then
12       v_0 := fire(F.low[u], svec);
13       v_1 := fire(F.high[u], svec);
14       res := MakeNode(B, F.var[u], v_0, v_1, max)
15     elsif F.var[u] = s then
16       if s ∈ •t \ t• then
17         v_0 := fire(F.high[u], tail(svec));
18         res := MakeNode(B, F.var[u], v_0, 0, max)
19       elsif s ∈ •t ∩ t• then
20         v_1 := fire(F.high[u], tail(svec));
21         res := MakeNode(B, F.var[u], 0, v_1, max)
22       else   /* s ∈ t• \ •t */
23         v_1 := fire(F.low[u], tail(svec));
24         res := MakeNode(B, F.var[u], 0, v_1, max)
25       fi
26     else   /* F.var[u] > s */
27       if s ∈ •t \ t• then
28         v_0 := fire(u, tail(svec));
29         res := MakeNode(B, s, v_0, 0, max)
30       else   /* s ∈ t• */
31         v_1 := fire(u, tail(svec));
32         res := MakeNode(B, s, 0, v_1, max)
33       fi
34     fi
35     ResultTable[u] := res;
36     return res
37   end fire;
38
39 begin
40   B.root := fire(F.root, (•t ∪ t•));
41   return B
42 end.
```

Der Einsatz solcher spezieller ROBDD-Operationen reduziert drastisch die Anzahl der insgesamt benötigten Knoten sowie auch den maximal benötigten Speicherplatz. Dies führt zu einer starken Steigerung der Performanz der Algorithmen (vgl. T. Yoneda u. a. [YHTM96], H. Ridder [Rid97], A. Noack [Noa99]).

Zero-suppressed BDDs

Betrachtet man die Zustandsvektoren eines Petrinetzes, so fällt auf, daß diese in der Regel in zweierlei Hinsicht dünn besetzt sind:

1. Zustandsvektoren enthalten viele Nullelemente, d. h. die Anzahl der Marken in einem Zustand ist viel kleiner als die Anzahl der Stellen.

2. Die Menge der von M_0 aus erreichbaren Markierungen $\mathcal{R}_{\mathcal{N}}(M_0)$ enthält nur einen Bruchteil der theoretisch möglichen $2^{|S|}$ Markierungen.

Diese Beobachtung ausnutzend hat S. Minato [Min93] eine ROBDD-Variante entwickelt, die sogenannten *Zero-suppressed binären Entscheidungsgraphen*, kurz ZBDDs. Sie unterscheiden sich von den ROBDDs durch eine speziell auf die Dünnbesetztheit zugeschnittene Eliminationsregel. Die Isomorphieregel bleibt dabei unverändert.

ZBDD-Eliminationsregel: Wenn die 1-Kante eines Knotens v auf den 0-Terminalknoten zeigt, dann eliminiere v und lenke alle in v eingehenden Kanten auf den 0-Nachfolgeknoten von v um.

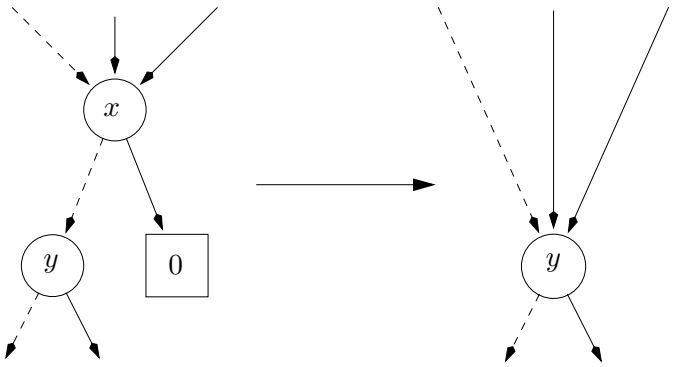


Abbildung 4.2: ZBDD-Eliminationsregel

Es werden also nur solche Variablen repräsentiert, die auch in einer Markierung vorkommen. Nichtauftreten einer Variablen x_i wird daher im Vergleich zu den ROBDDs unterschiedlich interpretiert:

ROBDD: Die dargestellte Funktion ist unabhängig von x_i.

ZBDD: $x_i = 1$ impliziert unmittelbar den Funktionswert 0.

Ist die Variablenanzahl und die Variablenordnung fest, so stellen ZBDDs ebenfalls eine kanonische Darstellungsform dar.
Die Abbildung 4.3 zeigt einen ROBDD und einen ZBDD, die dieselbe Zustandsmenge $\{\,(1,0,0,0),(0,1,0,0)\,\}$ repräsentieren. Die ZBDD-Darstellung benötigt nur vier Knoten im Gegensatz zu sieben Knoten der ROBDD-Darstellung. Man erkennt daran gut den Effekt der geänderten Eliminationsregel.

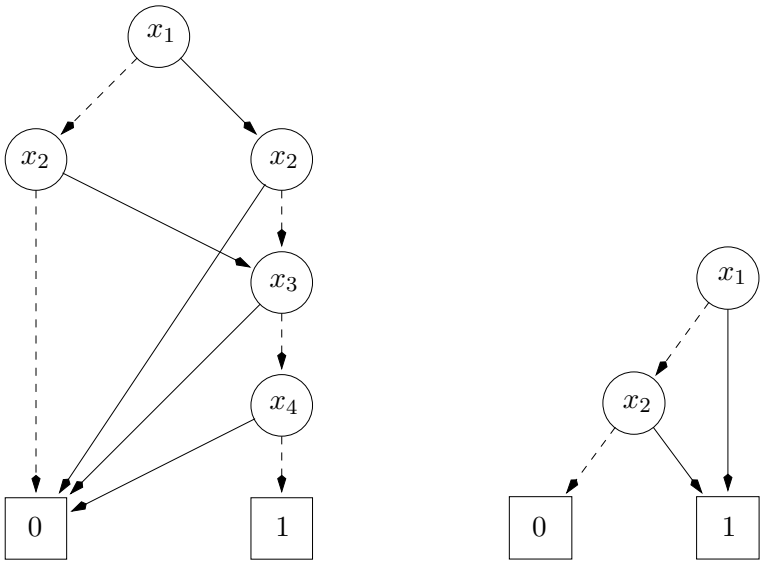


Abbildung 4.3: Zustandsmenge $\{\,(1,0,0,0),(0,1,0,0)\,\}$ in ROBDD- und ZBDD-Darstellung

Analog zu den ROBDDs können auch die ZBDDs aus den Grundgraphen durch Anwendung der entsprechenden Operationen erzeugt werden. Die dafür verwendeten Operationen unterscheiden sich natürlich von den ROBDD-Operationen.

empty()	Rückgabe: $\emptyset$
base()	Rückgabe: $\{\vec{0}\}$
univers(n)	Rückgabe: $2^{\{x_1,\ldots,x_n\}}$
subset1($\mathcal{M}, x_i$)	Rückgabe: die Teilmenge von $\mathcal{M}$ mit $x_i = 1$
subset0($\mathcal{M}, x_i$)	Rückgabe: die Teilmenge von $\mathcal{M}$ mit $x_i = 0$

change$(\mathcal{M}, x_i)$	Rückgabe: $\mathcal{M}$ mit invertierter Variablen x_i
union$(\mathcal{M}, \mathcal{M}')$	Rückgabe: $\mathcal{M} \cup \mathcal{M}'$
intsec$(\mathcal{M}, \mathcal{M}')$	Rückgabe: $\mathcal{M} \cap \mathcal{M}'$
diff$(\mathcal{M}, \mathcal{M}')$	Rückgabe: $\mathcal{M} \setminus \mathcal{M}'$
count$(\mathcal{M})$	Rückgabe: $\lvert\mathcal{M}\rvert$

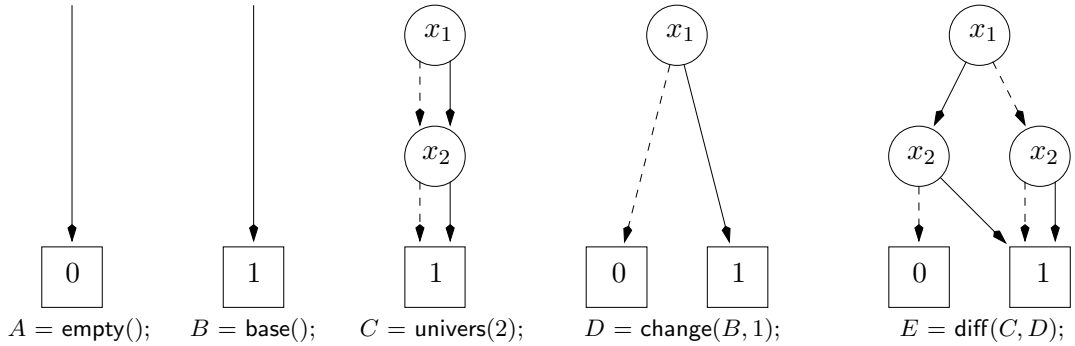


Abbildung 4.4: ZBDD-Operationen

Die Abbildung 4.4 zeigt die Konstruktion eines ZBDDs aus den Grundgraphen. Jede beliebige Markierung kann durch die base-Operation, gefolgt von change-Operationen der markierten Stellen erzeugt werden. Mit Hilfe der intsec-Operation kann dann überprüft werden, ob die konstruierte Markierung in einer Menge enthalten ist. Des weiteren erhält man das Komplement einer Markierungsmenge $\mathcal{M}$, indem man die Differenz zwischen der Potenzmenge der Stellen und $\mathcal{M}$ bildet.
Die Effizienz von ZBDDs für die Analyse von Petrinetzen belegen die Arbeiten von T. Yoneda u. a. [YHTM96] und A. Noack [Noa99]. Für die darin untersuchten Petrinetze ergibt sich eine starke Speicherplatzersparnis und damit eine große Geschwindigkeitszunahme.

Variablenordnung

Bekanntermaßen kann die Größe eines binären Entscheidungsgraphen stark von der Ordnung der Variablen abhängen (vgl. Abschnitt 3.2.6). Es gibt daher viele Ansätze [ATB94, SY96, Spr98], die versuchen, aus den strukturellen Eigenschaften des Modells statisch (d. h. einmalig vor Beginn der Zustandsmengenerzeugung) gute Variablenordnungen abzuleiten.

Im folgenden wollen wir zwei für Petrinetze geeignete Ansätze vorstellen, die gleichermaßen auf ROBDD- sowie auch auf ZBDD-Darstellungen anwendbar sind.

Der erste Ansatz basiert auf einer Arbeit von A. Noack [Noa99]. Es wird hierbei eine sehr einfache Heuristik unter Zuhilfenahme der Netzstruktur verwendet. Trotz des minimalen Aufwandes zur Erzeugung einer Variablenordnung erhält man mit diesem Ansatz sehr gute Ergebnisse.
Die angewandte Heuristik basiert auf einem Greedy-Algorithmus und bestimmt die Variablenordnung in der Reihenfolge von „unten“ (von den Terminalknoten) nach „oben“ (zu der Wurzel). Zur Bestimmung der nächsten Variablen aus der Menge der noch verbleibenden Variablen wird folgendes Maß verwendet:

$$\mathcal{M}(s) := \frac{\sum\limits_{t \in \bullet s} \frac{|\bullet t \cap S_a|}{|\bullet t|} + \sum\limits_{t \in s \bullet} \frac{|t \bullet \cap S_a|}{|t \bullet|}}{|\bullet s \cup s \bullet|}.$$

Die Stelle, und damit die ihr entsprechende Variable, mit dem bzgl. dieses Maßes höchsten Wertes wird als nächstes ausgewählt. Der Bezeichner S_a repräsentiert die Menge der bereits ausgewählten Stellen. Bei noch wenig ausgewählten Stellen erhält man ein definiertes Verhalten, wenn man im Falle, daß $|\bullet t \cap S_a|$ und $|t \bullet \cap S_a|$ den Wert 0 ergeben, dafür 0,1 einsetzt.
Für die Festlegung dieses Maßes wurde von den beiden folgenden Beobachtungen ausgegangen: Die Knotenzahl in den unteren Ebenen ist potentiell höher als in den oberen Ebenen und stark voneinander abhängige Variablen sollten in der Variablenordnung möglichst dicht beieinander liegen (vgl. Abschnitt 3.2.6).
In einem sicheren Petrinetz gilt folgender Zusammenhang für die Vor- und Nachstellen derselben Transition: Ist irgend eine Nachstelle (die nicht auch Vorstelle ist) markiert, so dürfen nicht alle Vorstellen belegt sein. Vor- und Nachstellen derselben Transition sind also voneinander abhängig. Die Formel verwendet diese generell geltende Markierungseinschränkung und erweitert diese, indem zusätzlich noch besonders Vor- bzw. Nachstellen von Transitionen mit einem möglichst hohen Anteil bereits ausgewählter Vor- bzw. Nachstellen bevorzugt werden.

Der zweite Ansatz basiert auf der Eigenschaft, daß eine Menge von Variablen eine BDD-Repräsentation mit linearer Größe bzgl. der Variablenanzahl besitzt, falls keine zwei Variablen dieser Menge gleichzeitig belegt sein können [SY96, Spr98]. Die Abbildung 4.5 verdeutlicht dies für die Variablenmenge $\mathcal{V} = \{ x_1, x_2, x_3, x_4 \}$. Sie stellt die möglichen Belegungen von $\mathcal{V}$, repräsentiert durch die Vektorenmenge $\mathcal{M} = \{ (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1) \}$, als ROBDD und als ZBDD dar. Eine ROBDD-Darstellung einer solchen Variablengruppe der Größe n hat $2n + 2$ Knoten. Eine ZBDD-Darstellung $n + 2$ Knoten. Weiterhin erkennt man anhand des Aufbaus der BDDs,

daß die Größe der BDD-Darstellung für solche Variablengruppen unabhängig von der Variablenordnung ist.

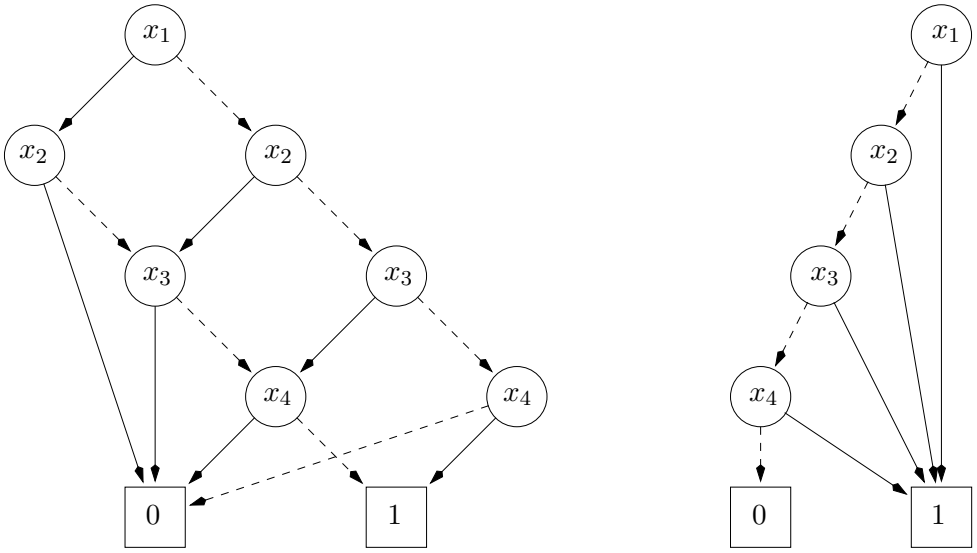


Abbildung 4.5: ROBDD- und ZBDD-Darstellung von $\mathcal{M}$

Diese Beobachtung führt zu einem heuristischen Verfahren, welches versucht, die Variablenmenge zur Darstellung von Petrinetzzuständen so zu partitionieren, daß möglichst große und ausgeglichene Variablenmengen mit obiger Eigenschaft entstehen. Wir bezeichnen im weiteren solche Variablenmengen als *Variablengruppen*. Die Darstellung der Markierungen eines Petrinetzes basiert auf einer bijektiven Abbildung zwischen der Menge der Stellen des Petrinetzes und den Variablen des entsprechenden BDDs (vgl. Abschnitt 4.1).
In der Arbeit [Spr98] dienen daher statisch berechnete 1-S-Invarianten als Ausgangspunkt einer Partitionierung. Das dort verwendete heuristische Verfahren ist im Algorithmus 9 (Gruppierung) dargestellt. Der Algorithmus ordnet nacheinander die Variablen den schon existierenden Variablengruppen zu, so daß keine Variablen in einer Variablengruppe zusammengefaßt werden, für die nicht eine gemeinsame 1-S-Invariante existiert. Die Variablen werden vorher aufsteigend, bzgl. der Anzahl der Zugehörigkeit zu verschiedenen 1-S-Invarianten, geordnet. Daraus ergibt sich eine ausgewogenere Anzahl von Variablen pro Variablengruppe und damit eine insgesamt kleinere BDD-Darstellung.
Die BDD-Größe läßt sich durch eine gut gewählte Anordnung der Variablengruppen noch weiter verkleinern. Dies erreicht man durch ein weiteres heuristisches Verfahren, welches den *Grad der Abhängigkeit* zwischen den Variablengruppen zur Anordnung nutzt. Der Grad der Abbhängigkeit zweier Variablengruppen g_i, g_j wird durch die

Algorithmus 9 (Gruppierung von Variablen)

```
1  Gruppierung([S, T, F, M_0]) ≡
2    G := ∅;
3    I := ∅;
4    svec := ();
5
6    func clustering(svec) ≡
7      while svec ≠ () do
8        s := head(svec);
9        svec := tail(svec);
10       /* suche eine Variablengruppe g ∈ G so daß gilt: ∀s' ∈ g ∃i ∈ I mit s, s' ∈ i */
11       g := search(G);
12       if g ≠ ∅ then
13         g := g ∪ {s}
14       else
15         G := G ∪ {{s}}
16       fi
17     od;
17     return svec
18   end clustering;
19
20 begin
21   I := 1-S-Invarianten([S, T, F, M_0]);
22   /* ordne Stellen aufsteigend bzgl. der Anzahl von zugehörigen 1-S-Invarianten */
23   svec := order(S);
24   return clustering(svec)
25 end.
```

Anzahl von Variablenpaaren s_i, s_j mit $s_i \in g_i$ und $s_j \in g_j$ ermittelt, für die eine 1-S-Invariante existiert, die beide Stellen enthält. Dies läßt sich in einem Abhängigkeitsgraph darstellen, dessen Knoten die Variablengruppen sind, und die gewichteten Kanten den Grad der Abhängigkeit zwischen den Variablen repräsentieren.

Basierend auf diesem Abhängigkeitsmaß ordnet der Algorithmus 10 (Gruppenordnung) die Variablengruppen unter Verwendung dieses Abhängigkeitsgraphen. Es werden nacheinander die Variablengruppen ausgewählt, die die größte Summe von Abhängigkeiten zu schon ausgewählten Variablengruppen haben.

Um die Variablenordnung weiter zu verbessern, werden die Variablen innerhalb einer Variablengruppe in Abhängigkeit zu der Anzahl von gemeinsamen 1-S-Invarianten mit den Variablen der beiden Nachbarvariablenordnungen angeordnet.

Variablen von Stellen, die in keiner 1-S-Invarianten vorkommen, werden zur Vervollständigung der Codierung an die schon existierenden Variablen angefügt.

Algorithmus 10 (Anordnung von Variablengruppen)

```
1  Gruppenordnung([S, T, F, M_0]) ≡
2    G := ∅;
3    I := ∅;
4    L := ();  /* Vektor der ausgewählten Variablengruppen */
5
6    func ordering(G) ≡
7      /* wähle Variablengruppe mit der größten Kantengewichtssumme */
8      g := select(G, ());
9      G := G \ g;
10     L := append(L, g);
11     while G ≠ ∅ do
12       /* wähle Variablengruppe mit der größten Kantengewichtssumme
13          bzgl. schon ausgewählter Variablengruppen */
14       g := select(G, L);
15       G := G \ g;
16       L := append(L, g)
17     od;
17     return L
18   end ordering;
19
20 begin
21   I := 1-S-Invarianten([S, T, F, M_0]);
22   G := compute_dependabilitygraph(Gruppierung([S, T, F, M_0]), I);
23   return ordering(G, L)
24 end.
```

Kompakte Codierung von Markierungen

Neben der Variablenordnung hat auch die für die Codierung einer Markierung benötigte Variablenanzahl einen großen Einfluß auf die Größe der entsprechenden BDD-Darstellung. Im traditionellen Ansatz (vgl. Abschnitt 4.1) wird jeder Stelle genau eine Variable zugeordnet. Mit diesem Ansatz ist es potentiell möglich, $2^{|S|}$ Markierungen zu codieren, was aber nur in den seltensten Fällen ausgenutzt wird. Meist wird zur Darstellung der von M_0 aus erreichbaren Markierungen eines Petrinetzes $\mathcal{R}_{\mathcal{N}}(M_0)$ nur ein Bruchteil dieser Codierungsmöglichkeiten tatsächlich benötigt.
Die Ansätze [Spr98, PC98] benutzen daher, basierend auf strukturellen Informationen der zu analysierenden Petrinetze, eine kompaktere (d. h. mit weniger Variablen auskommende) Codierung der Markierungen.
Der Ansatz [Spr98] basiert auf den zu einem Petrinetz gehörenden 1-S-Invarianten. Bei einer 1-S-Invarianten wird durch die Angabe der markierten Stelle auch die Belegung der anderen Stellen der 1-S-Invariante bestimmt. Numeriert man also die Stellen einer 1-S-Invarianten i durch und codiert diesen Wert anschließend binär, so genügen $\lceil \log_2 |i| \rceil$ Variablen zur Codierung.

Beispiel 8 (1-S-Invariante $i = \{ s_1, s_2, s_3, s_4 \}$)

Stelle	traditionell	binär
s_1	0001	00
s_2	0010	01
s_3	0100	10
s_4	1000	11

Zur kompakten Codierung von Markierungen benötigt man daher eine möglichst vollständige Überdeckung der Stellen eines Petrinetzes mit dessen 1-S-Invarianten. Nicht überdeckte Stellen werden weiterhin traditionell codiert. Die 1-S-Invarianten des Petrinetzes aus Abbildung 2.1 ergeben die folgende Überdeckung.

$$\{ \{s_1, s_2, s_4\}, \{s_1, s_3, s_5\}, \{s_6\} \}$$

Codiert man jede dieser 1-S-Invarianten binär, so benötigt man anstatt der ursprünglichen sechs Variablen nur fünf.
Überdeckungen führen normalerweise zu einigen Redundanzen in der Codierung von Stellen. In unserem Beispiel ist die Stelle s_1 in zwei 1-S-Invarianten enthalten und wird damit auch zweimal codiert. Verhindert man das mehrfache Auftreten von Stellen, indem man beim Hinzufügen neuer 1-S-Invarianten schon von anderen 1-S-Invarianten überdeckte Stelle aus ihnen entfernt, erhält man eine Partitionierung der Stellen in Variablengruppen (vgl. vorherigen Abschnitt).

$$\{ \{s_1, s_2, s_4\}, \{s_3, s_5\}, \{s_6\} \}$$

Im Gegensatz zur Codierung von 1-S-Invarianten stößt man beim Codieren von Variablengruppen auf das Problem, daß diese unmarkiert sein können. Daher ist eine spezielle Behandlung (Codierung) der Unmarkiertheit von Variablengruppen erforderlich.
Man kann diesbezüglich die folgenden drei Variablengruppenvarianten unterscheiden:

1. Variablengruppen, die 1-S-Invarianten sind.
   Diese Variablengruppen sind nie unmarkiert und benötigen daher auch keine entsprechende Codierung.

2. Variablengruppen, die keine 1-S-Invarianten sind und deren Stellenanzahl keine 2er-Potenz ist.
   Bei der binären Codierung der Stellen solcher Variablengruppen bleibt mindestens eine Codierung ungenutzt. Man kann daher immer eine freie Codierung zur Repräsentation der Unmarkiertheit auswählen.

3. Variablengruppen, die keine 1-S-Invarianten sind und deren Stellenanzahl eine 2er-Potenz ist.
   Durch Hinzufügen einer zusätzlichen Variablen für jede solche Variablengruppe erhält man eine einfache Möglichkeit, deren Unmarkiertheit zu codieren. Dies hat den ungewollten Effekt, die zur Codierung von Markierungen benötigte Variablenanzahl zu erhöhen, und ist daher nur beim Vorhandensein einer kleinen Anzahl solcher Variablengruppen akzeptierbar.

   Eine andere Möglichkeit, die ohne zusätzliche Variablen auskommt, basiert auf einer Doppelbedeutung der Codierung einer ausgewählten Stelle jeder solchen Variablengruppe. Die Codierung der ausgewählten Stelle hat dabei einerseits die Bedeutung, daß die Stelle markiert ist, und kann andererseits die Unmarkiertheit der ganzen Variablengruppe bedeuten. Um dies unterscheiden zu können, muß jede Transition, die eine solche Stelle im Vorbereich hat, vor ihrem Schalten die Markierungen herausfiltern, bei denen keine weitere Stelle aus der zu der Variablengruppe gehörigen 1-S-Invariante markiert ist. Man ordnet dazu jeder Transition eine Filterfunktion zu und führt diese vor dem jeweiligen Schalten der Transitionen aus. Dieser Ansatz benötigt zwar keine zusätzlichen Variablen, erhöht aber die Komplexität des Schaltvorgangs. Da die Komplexität von BDD-Operationen aber von der BDD-Größe maßgeblich bestimmt wird, ist eine möglichst kompakte Codierung trotz des Mehraufwandes vorzuziehen.

Diese Überlegungen waren die Grundlagen der Arbeit [Spr98]. Die dort weiterentwickelten Ideen einer kompakten Codierung von Markierungen wollen wir im folgenden näher betrachten.
Da der Aufwand, die Stellen eines Petrinetzes so in Variablengruppen zu unterteilen, daß die Gesamtanzahl der zur Codierung benötigten Variablen minimal ist, exponentiell mit der Anzahl der Stellen zunimmt, wird ein heuristisches Verfahren verwendet. Der Algorithmus 11 (Partitionierung) stellt ein solches Verfahren dar. Die Auswahl der Variablengruppen erfolgt nacheinander unter dem Gesichtspunkt der von einer Variablengruppe in den verbliebenen Variablengruppen überdeckten Stellenanzahl (select_max_place_cover). Dabei werden Stellen, die in unterschiedlichen Variablengruppen vorkommen, mehrfach gezählt. Man versucht dadurch, die noch verbleibenden Variablengruppen, und damit die Anzahl der zur Codierung benötigten Variablen, möglichst klein zu bekommen.
Die durch den Algorithmus 11 (Partitionierung) erzeugten Variablengruppen G werden zu einer kompakten Codierung der Markierungen verwendet. Dazu ordnet man jeder Variablengruppe $g \in G$ eine unterschiedliche Menge von Variablen $V(g)$ zu. Daraus ergibt sich die Menge der zur Codierung einer Markierung notwendigen Variablen $V_{\mathcal{N}}$ als

$$V_{\mathcal{N}} := \bigcup_{g_i \in G} V(g_i).$$

Algorithmus 11 (Unterteilung der Stellen in Variablengruppen)

```
1  Partitionierung([S, T, F, M_0]) ≡
2    G := ∅;  /* Vektor der noch verbleibenden Variablengruppen */
3    L := ();  /* Vektor der ausgewählten Variablengruppen */
4
5    func partition(G, L) ≡
6      /* vollständige Überdeckung von S */
7      forall s ∈ S do
8        forall g ∈ G do
9          if s ∉ g then G := G ∪ { {s} } fi
10       od
11     od;
12     while G ≠ ∅ do
13       g := select_max_place_cover(G);
14       G := G \ g;
15       L := append(L, g);
16       forall g' ∈ G do
17         G := G \ g';
18         G := G ∪ { g' \ g }
19       od
20     od;
21     return L
22   end partition;
23
24 begin
25   G := 1-S-Invarianten([S, T, F, M_0]);
26   return partition(G, L)
27 end.
```

Wir bezeichnen im weiteren die Literale zu den Variablen $V_{\mathcal{N}}$ mit L. Zur weiteren Beschreibung der Codierung von Markierungen und der Schaltdynamik von Petrinetzen definieren wir:

- Die Abbildung $\mathcal{G}\colon S \longrightarrow G$, die jeder Stelle s die zugehörige Variablengruppe $g \in G$ zuordnet.
- Die Abbildung $\mathcal{I}\colon G \longrightarrow 2^S$, die jeder Variablengruppe $g \in G$, sofern vorhanden, die 1-S-Invariante zuordnet, aus der g durch den Algorithmus 11 (Partitionierung) entstanden ist. Ansonsten wird die leere Menge zurückgegeben.
- Die Abbildung $\mathcal{C}'\colon S \longrightarrow 2^L$, die zu jeder Stelle s ein Monom aus Literalen zu den Variablen $V(\mathcal{G}(s))$ zurückgibt. Das Monom spiegelt dabei die eindeutige binäre Codierung der Stelle s wieder. Für die Stelle $s_3 \in i$ aus Beispiel 8 ergibt sich mit $V(i) := \{\, x_5, x_6 \,\}$ die binäre Codierung $\mathcal{C}'(s_3) = \{\, x_5, \overline{x_6} \,\}$.
- Die Abbildung $\mathcal{E}\colon S \longrightarrow 2^L$, die jeder Stelle s das Monom, bestehend aus den Komplementen der Variablen $V(\mathcal{G}(s))$, zuordnet. Diese Abbildung definiert die Unmarkiertheit einer Variablengruppe durch eine 0-Belegung ihrer Variablen. Dies geschieht, aus Einfachheitsgründen, einheitlich für alle Variablengruppen, d. h. auch für 1-S-Invarianten. Die Unmarkiertheit der Variablengruppe i aus Beispiel 8 ergibt sich als $\mathcal{E}'(s) = \{\, \overline{x_5}, \overline{x_6} \,\}$ mit $s \in i$. Die dabei auftretenden Doppelbedeutungen mit Stellencodierungen werden durch die als nächstes definierten jeweiligen Filterfunktionen aufgelöst.
- Die Abbildung $\mathcal{F}\colon S \longrightarrow \chi_{\mathcal{M}}$ liefert zu jeder Stelle s die charakteristische Funktion der kompakten Codierung aller Markierungen $\mathcal{M}$, bei denen s „wirklich“ markiert ist. Dies geschieht durch Betrachtung der Markiertheit der Stellen, die in der zu $\mathcal{G}(s)$ gehörigen 1-S-Invarianten liegen. Sei $\{\, s_1, \ldots, s_k \,\} := \mathcal{I}(\mathcal{G}(s)) \backslash \mathcal{G}(s)$, dann ist die Abbildung $\mathcal{F}$ wie folgt definiert:

$$\mathcal{F}(s) := \begin{cases} \prod\limits_{s_i \in \mathcal{C}'(s)} s_i, & \text{falls } \mathcal{C}'(s) \neq \mathcal{E}(s) \\ \prod\limits_{s_i \in \mathcal{C}'(s)} s_i \wedge \neg\mathcal{F}(s_1) \wedge \cdots \wedge \neg\mathcal{F}(s_k), & \text{sonst.} \end{cases}$$

  Aus der Abbildung $\mathcal{F}$ werden für jede Transition $t \in T$ Filter erzeugt, die nur Markierungen durchlassen, bei denen die Stellen aus dem Vorbereich von t auch wirklich markiert sind. Dadurch werden die möglichen Doppeldeutigkeiten bei der Codierung aufgelöst. Die Terminierung der rekursive Definition von $\mathcal{F}$ ist durch die Art der Konstruktion der Variablengruppen gegeben.

Die konkrete Codierung einer Stelle s hängt von der Art ihrer zugehörigen Variablengruppe $\mathcal{G}(s)$ ab. Wir unterscheiden diesbezüglich sechs Variablengruppenvarianten.

1. Variablengruppen $g \in G$ mit $|g| = 1$ und die Stelle $s \in g$ ist in keiner 1-S-Invarianten enthalten. Die Stelle s wird traditionell durch eine eigene Variable codiert. Für die Stelle gilt also $\mathcal{C}'(s) = \{x_i\}$ und $\mathcal{E}(s) = \{\overline{x_i}\}$.

2. Variablengruppen $g \in G$ mit $|g| = 1$ und g ist eine 1-S-Invariante. Die Stelle $s \in g$ ist unter jeder erreichbaren Markierung markiert. Sie wird daher durch keine eigene Variable codiert, sondern wird immer implizit als markiert angenommen.

3. Variablengruppen $g \in G$ mit $|g| = 1$ und g ist enthalten in einer 1-S-Invariante. Die Stelle $s \in g$ wird nicht durch eine eigene Variable codiert, sondern nur indirekt durch die Belegung der Stellen der zu g gehörigen 1-S-Invarianten $\mathcal{I}(g)$.

4. Variablengruppen $g \in G$ mit $|g| > 1$ und g ist eine 1-S-Invariante. Die Stellen von g werden binär mit $\lceil \log_2 |g| \rceil$ Variablen codiert.

5. Variablengruppen $g \in G$ mit $|g|$ ist keine 2er-Potenz und g ist keine 1-S-Invariante. Die Stellen von g werden binär mit $\lceil \log_2 |g| \rceil$ Variablen codiert, wobei keine der Stellen mit dem Monom bestehend aus den Komplementen der Variablen $V(g)$ codiert wird.

6. Variablengruppen $g \in G$ mit $|g|$ ist eine 2er-Potenz und g ist keine 1-S-Invariante. Die Stellen von g werden binär mit $\lceil \log_2 |g| \rceil$ Variablen codiert.

Analog zum traditionellen Codierungsansatz (vgl. Abschnitt 4.1) definieren wir eine bijektive Abbildung $\mathcal{C}\colon 2^S \longrightarrow \mathbb{B}^{|V_\mathcal{N}|}$, $M \mapsto (x_1, \ldots, x_{|V_\mathcal{N}|})$ mit

$$x_i := \begin{cases} 1, & \text{falls das Literal } x_i \in \bigcup\limits_{s_i \in M} \mathcal{C}'(s_i) \\ 0, & \text{sonst} \end{cases}$$

die jede Markierung $M \in 2^S$ eindeutig durch einen Vektor aus $\mathbb{B}^{|V_\mathcal{N}|}$ darstellt. Die Abbildung $\mathcal{C}$ ist bijektiv, wodurch jeder Markierung genau eine Variablenbelegung zugeordnet wird. Dadurch bleibt der einfache Äquivalenztest zweier Markierungsmengen auf ROBDD-Basis erhalten. Des weiteren läßt sich durch die neue Codierungsfunktion $\mathcal{C}$, analog zum Abschnitt 4.1, die charakteristische Funktion $\chi_\mathcal{M}$ einer Menge von Markierungen in kompakter Codierung definieren.

Die das Schalten von Transitionen beschreibenden Schaltfunktionen $post_t$ lassen sich damit folgendermaßen definieren:

$$
\begin{aligned}
filter_t(\chi_{\mathcal{M}}) &:= \chi_{\mathcal{M}} \wedge \bigwedge_{s_i \in \bullet t} \mathcal{F}(s_i) \\
post'_t(\chi_{\mathcal{M}}) &:= \left[\chi_{\mathcal{M}}\left[\bigcup_{s_i \in \bullet t} \mathcal{C}'(s_i) \right] \wedge \bigwedge_{s_i \in \bullet t} \mathcal{E}(s_i) \right]_{\left[\bigcup_{s_i \in t\bullet} \mathcal{E}(s_i) \right]} \wedge \bigwedge_{s_i \in t\bullet} \mathcal{C}'(s_i) \\
post_t(\chi_{\mathcal{M}}) &:= post'_t(filter_t(\chi_{\mathcal{M}}))
\end{aligned}
$$

Die Schaltfunktionen $post_t$ bestehen aus einem Filter, der die möglichen Doppelbedeutungen einzelner Stellen durch Herausfiltern von unzulässigen Markierungen auflöst. Das eigentliche Schalten der Transitionen findet in zwei Schritten in den Funktionen $post'_t$ statt. Im ersten Schritt werden die Markierungen, die alle Vorstellen der zu schaltenden Transition markiert haben, aussortiert und die zu den Stellen gehörigen Variablengruppen als unmarkiert gekennzeichnet. Im zweiten Schritt werden die Markierungen aussortiert, deren den Nachstellen zugehörigen Variablengruppen unmarkiert sind und entsprechend der Nachstellen neu markiert.
Der Algorithmus 7 (ReachableSet) zur Erzeugung aller erreichbaren Markierungen eines Petrinetzes muß für die kompakte Codierung folgendermaßen modifiziert werden.

```
7  forall  t ∈ T  do                    7  forall  t ∈ T  do
8    New := New ∪ post_t(Old)    ⟹     8    New := New ∪ post'_t(filter_t(Old))
9  od                                   9  od
```

Die in der Arbeit [Spr98] verwendeten Anwendungsbeispiele kamen im Schnitt auf eine 50%ige Reduktion der zur Codierung benötigten Variablen. Daraus ergab sich eine große Speicherplatzersparnis und eine um Größenordnungen höhere Abarbeitungsgeschwindigkeit. Der durch die kompakte Codierung erzeugte Mehraufwand wurde in den getesteten Beispielen durch die sich ergebenden Einsparungen bei weitem aufgehoben.

4.3 Zusammenfassende Bemerkungen

Der fundamentale Isomorphismus zwischen einer endlichen Booleschen Algebra und der entsprechenden Mengenalgebra ermöglicht es, Markierungsmengen durch ihre charakteristische Funktion als ROBDD darzustellen. Man erhält dadurch eine kompakte und effiziente Darstellungsform für Markierungsmengen, die uns gleichzeitig schnelle Operationen zur Verarbeitung bietet.
Auch das dynamische Verhalten von Petrinetzen ist damit auf Boolesche Operationen abbildbar. Die sich ergebenden, auf Fixpunktiteration basierenden Vorwärtsiterations- und Rückwärtsiterationsverfahren ermöglichen es, Aussagen über die Erreichbarkeit

von Markierungen zu machen. Dadurch sind sogenannte Sicherheitseigenschaften für Petrinetze überprüfbar.
Das Standardfixpunktverfahren bietet eine Vielzahl von Optimierungsmöglichkeiten, mit denen sich seine Effizienz um Größenordnungen verbessern läßt (vgl. z. B. [Noa99]). Als sehr effektiv erweisen sich dabei solche Verfahren, die Petrinetzstrukturinformationen zur Reduktion der ROBDD-Größe ausnützen. Dabei dienen die Strukturinformationen einerseits zur Bestimmung guter Variablenordnungen und andererseits liefern sie Ansätze die zur Codierung von Zuständen benötigte Variablenanzahl zu reduzieren (vgl. [Spr98]).

5 Temporale Logik

Die Verifikation von software-basierten Systemen ist wünschenswert, aber meist ein sehr schwieriges und aufwendiges Unterfangen. Die durch das Konzept der Nebenläufigkeit bedingte Komplexität potenziert die Notwendigkeit einer Verifikation sowie deren Schwierigkeit. Nichtdestotrotz haben sich über die Jahre relativ erfolgreiche Verifikationsverfahren [CW96] etabliert. Diese Ansätze beziehen sich meist auf eine spezielle aber sehr wichtige Klasse von nebenläufigen Systemen, die sogenannten *reaktiven Systeme* [MP92].
Reaktive Systeme zeichnen sich durch eine fortlaufende Interaktion mit ihrer Umgebung aus, wobei im allgemeinen nur geringe Datenmengen manipuliert werden. Des weiteren handelt es sich oft um Systeme mit endlich vielen Zuständen oder um Systeme, die sich sinnvoll auf eine endliche Zustandmenge abstrahieren lassen. Beispiele für klassische reaktive Systeme sind nebenläufige Kommunikationsprotokolle, Prozeßkontrollsysteme und interaktive Programme. Ihr Einsatz erfolgt häufig in sicherheitskritischen Bereichen, wie z. B. Flugzeugen, Atomkraftwerken und Steuerungen medizinischer Geräte, so daß sie eine Klasse von Systemen mit hohem Verifikationsbedarf darstellen. Eine solche Verifikation ist auf Grund ihrer speziellen Struktur auch meist mit vertretbarem Aufwand möglich.
Die im Kapitel 2 eingeführten sicheren Stellen/Transitions-Netze eignen sich besonders zur Modellierung von Systemabläufen mit geringen Datenmanipulationen und werden daher von uns zur Beschreibung reaktiver Systeme eingesetzt. Im weiteren Verlauf wollen wir nicht mehr zwischen dem eigentlichen System und seinem Petrinetzmodell unterscheiden.
Als Verifikation von reaktiven Systemen wird das formal mathematische Überprüfen von in einer logischen Sprache beschriebenen Eigenschaften (einer Spezifikation) bezeichnet. Für reaktive Systeme sind dabei vor allem Aussagen über die Zustände des Systems und deren „zeitlichem Zusammenhang“ von Interesse. Klassisch unterteilt man die zu überprüfenden Eigenschaften in die folgenden zwei Kategorien [Lam77]:

1. *Sicherheitseigenschaften*: Das System gerät niemals in einen unerwünschten Zustand.

2. *Lebendigkeitseigenschaften*: Das System erreicht letztendlich einen erwünschten Zustand.

Systemzustände besitzen eine endliche Anzahl von „Informationen“, die durch sogenannte *elementare Aussagen (atomic propositions)* beschreibbar sind. Daher können wir die klassische Aussagenlogik zur Beschreibung von Zuständen benützen. In unserem Fall entspricht ein Systemzustand gerade einer Markierung der Stellen eines Petrinetzes. Die elementaren Aussagen sind daher Aussagen über die Markiertheit bzw. Unmarkiertheit von Stellen.
Die Wahrheitswerte elementarer Aussagen verändern sich aber meist im „Laufe der Zeit“ (d. h. durch den Wechsel des Systemzustandes), was nicht mehr durch die Aussagenlogik beschreibbar ist.
Um also Aussagen über die zeitliche Entwicklung eines Systems bzw. über die zeitlichen Bezüge zwischen Systemzuständen machen zu können, wird die Aussagenlogik um sogenannte Temporaloperatoren angereichert. Man erhält dadurch die sogenannte *temporale Logik* [Pnu77, Eme90].
Im weiteren wollen wir uns mit einer speziellen temporalen Logik, der *Linear-time Temporal Logic* (kurz LTL) [Pnu80, LP85], näher beschäftigen. Ein nebenläufiges System stellt sich für die LTL als eine Menge von Systemabläufen dar, wobei jeder Systemablauf unabhängig von allen anderen Abläufen betrachtet wird. Der in der LTL verwendete Zeitbegriff basiert auf „Zeitpunkten“ (den Zuständen eines Systemablaufes), die durch eine Relation der direkten Nachfolge (Erreichbarkeitsrelation) verbunden sind. Der Ablauf der Zeit erfolgt linear, da zu einem Systemzustand immer genau ein Nachfolgezustand existiert, und in diskreten Schritten.
Das Verfahren zur Beantwortung der Frage „Erfüllt ein gegebenes nebenläufiges System $\mathcal{N}$ die gegebene LTL-Spezifikation ϕ?“ wird als *Modelchecking* bezeichnet.
Für reaktive Systeme mit endlich vielen Zuständen läßt sich dafür ein effizientes Verfahren angeben. Als Grundlage dieses Verfahrens wird meist auf die Automatentheorie zurückgegriffen [VW86, Var96]. Der Zusammenhang zwischen der LTL und der Automatentheorie basiert auf der Tatsache, daß jeder Ausführungspfad eines nebenläufigen Systems als unendliche Sequenz von Zuständen betrachtet werden kann. Da jeder Zustand vollständig durch eine endliche Menge elementarer Aussagen beschreibbar ist, kann ein Ausführungspfad auch als unendliches Wort über dem Alphabet der Wahrheitswertzuweisungen der elementaren Aussagen betrachtet werden. Ein solches System ist also ein aus endlich vielen Zuständen bestehender Generator für unendliche Worte. Andererseits läßt sich aus jeder LTL-Formel ein endlicher Automat über unendlichen Worten konstruieren, der genau die Zustandssequenzen akzeptiert, die die Formel erfüllen. Er spiegelt einen aus endlich vielen Zuständen bestehenden Akzeptor für unendliche Wörter wider. Durch diese Betrachtungsweise reduziert sich das Verifikationsproblem auf das rein automatentheoretische Problem: „Überprüfe, ob die Sprache $L(\mathcal{N})$ eine Teilmenge der Sprache $L(\phi)$ ist!“. Hierbei bezeichnet $L(\mathcal{N})$ die durch das System $\mathcal{N}$ erzeugte Sprache und $L(\phi)$ die durch LTL-Formel ϕ akzeptierte Sprache.
Die algorithmische Natur dieses Ansatzes macht den Modelcheckingprozeß vollständig

automatisierbar. Im folgenden gehen wir auf die wesentlichen Punkte dieses Ansatzes ein.

5.1 Transitionssysteme

Als Interpretationsgrundlage der von uns betrachteten temporallogischen Formeln verwenden wir Transitionssysteme.

Definition 32 (Transitionssystem)
Ein *Transitionssystem* ist ein Tupel $\mathcal{T} = [Q, \longrightarrow, q_0]$ mit

1. Q ist eine endliche Menge von Zuständen,
2. $\longrightarrow \subseteq Q \times Q$ ist eine binäre Relation mit $\forall q \in Q\ \exists q' \in Q :\ q \longrightarrow q'$ (d. h. $\longrightarrow$ ist total) und
3. $q_0 \in Q$ ist der Initialzustand.

Man kann ein Transitionssystem auch als eine Menge von „Zeitpunkten“ interpretieren, die durch eine Relation der direkten Nachfolge verbunden sind. Jeder Zeitpunkt hat dabei immer mindestens einen Nachfolger, so daß die „Zeit“ nicht stehenbleiben kann. Das Verhalten eines Transitionssystems ist durch die bei q_0 beginnenden Abfolgen von Systemzuständen beschreibbar. Solche Systemabläufe werden meist auch als *Wege* oder *Pfade* bezeichnet.

Definition 33 (Weg, Pfad)
Ein *Weg* oder *Pfad* durch ein Transitionssystem $\mathcal{T}$ ist eine bei q_0 beginnende unendliche Folge $\pi = q_0 q_1 q_2 \ldots$ von Systemzuständen ($q_i \in Q, i \in \mathbb{N}$), für die

$$q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow \cdots$$

gilt.

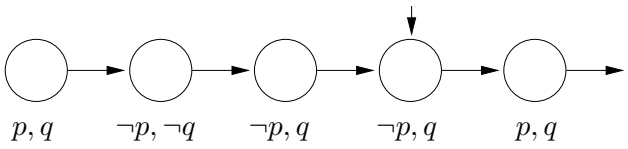
Die Menge aller Wege eines Transitionssystems $\mathcal{T}$ wird auch als Sprache $L(\mathcal{T})$ von $\mathcal{T}$ bezeichnet.
Betrachtet man die Interleaving-Semantik eines Petrinetzes $\mathcal{N}$, so läßt sich dessen Erreichbarkeitsgraph $\mathcal{RG}(\mathcal{N}) = [\mathcal{R}_\mathcal{N}(M_0), \mathcal{K}_\mathcal{N}]$ als Transitionssystem $\mathcal{T}_\mathcal{N} = [\mathcal{R}_\mathcal{N}(M_0), \mathcal{K}'_\mathcal{N}, M_0]$ interpretieren. Hierbei verzichtet man auf die Beschriftung der Kanten und definiert $\mathcal{K}'_\mathcal{N} := \{\, [M, M'] \mid [M, t, M'] \in \mathcal{K}_\mathcal{N} \,\}$. Damit die Relation total wird (d. h. die Zeit nicht stehen bleibt), hängen wir an jede tote Markierung des Petrinetzes eine Schleife zu sich selber an.
Bei dieser Transformation in ein Transitionssystem geht die Information verloren, welche Transition für welchen Zustandsübergang verantwortlich ist. Was übrig bleibt ist die Information über die Stellenbelegungen, die in jedem Zustand codiert ist.
Man definiert daher die Sprache $L(\mathcal{N})$ eines Petrinetzes $\mathcal{N}$ als die Sprache seines entsprechenden Transitionssystems $L(\mathcal{T}_\mathcal{N})$.

5.2 Linear-time Temporal Logic

LTL ist eine Sprache zur Beschreibung von Eigenschaften unendlicher Zustandsabfolgen. Dazu wird den Abfolgezuständen a priori eine Menge von elementaren Aussagen zugeordnet. Eine LTL-Formel ist eine mit Hilfe von aussagenlogischen Verknüpfungen aus den elementaren Aussagen und den sogenannten *temporallogischen Operatoren* zusammengesetzte Aussage.
Die Interpretation einer LTL-Formel erfolgt über den den Zuständen einer Abfolge zugeordneten elementaren Aussagen. Die temporallogischen Operatoren bestimmen dabei, in welchem Zustand der Abfolge die LTL-Formel (bzw. Teile davon) interpretiert werden. Die zu der LTL gehörigen temporallogischen Operatoren sind:

- $\mathsf{X}\,\phi$ (gesprochen: *nexttime* ϕ): Im unmittelbar nächsten Zustand der Abfolge gilt ϕ.
- $\mathsf{F}\,\phi$ (gesprochen: *finally* ϕ oder *eventually* ϕ): Letztendlich ist ein Zustand der Abfolge erreichbar, in dem ϕ gilt.
- $\mathsf{G}\,\phi$ (gesprochen: *globally* ϕ oder *always* ϕ): Von jetzt ab und in allen zukünftigen Zuständen der Abfolge gilt ϕ.
- $\phi_1\,\mathsf{U}\,\phi_2$ (gesprochen: ϕ_1 *until* ϕ_2): Irgendwann ist ein Zustand der Abfolge erreichbar, in dem ϕ_2 gilt. Bis dahin erfüllen alle durchlaufenen Zustände ϕ_1.
- $\phi_1\,\mathsf{R}\,\phi_2$ (gesprochen: ϕ_1 *releases* ϕ_2): Existiert ein zukünftiger Zustand in der Abfolge, für den ϕ_2 nicht gilt, so existiert ein bis dahin durchlaufener Zustand, in dem ϕ_1 gilt.

Beispiel 9 (Interpretation von LTL-Formeln)
Systemablauf:



In dem aktuellen Zustand des Systemablaufes (gekennzeichnet durch eine zusätzliche Eingangskante) gilt:

- Die Formeln $\neg p \wedge q$ und $\mathsf{X}\,(p \wedge q)$ sind wahr.
- Die Formel $\mathsf{G}\,p$ ist falsch.

5.2.1 Formale Definition

Sei Σ ein endliches Alphabet und Π eine Menge von Aussagen über Σ, d. h. eine Menge von Abbildungen $\{ \pi_i \mid \pi_i \colon \Sigma \longrightarrow \mathbb{B} \}$. Die Abbildungen $true \colon \Sigma \longrightarrow \mathbb{B}$, $\sigma \mapsto 1$ und $false \colon \Sigma \longrightarrow \mathbb{B}$, $\sigma \mapsto 0$ werden dabei immer implizit zu Π gehörig betrachtet.
Wie üblich bezeichnen wir mit Σ^ω die Menge aller unendlichen Wörter über dem Alphabet Σ.

Definition 34 (Syntax von LTL)
Die Menge der LTL-Formeln über den Aussagen Π ist induktiv definiert durch:

1. $true$ und $false$ sind LTL-Formeln.
2. Ist $\phi \in \Pi$, so sind ϕ und $\neg\phi$ LTL-Formeln.
3. Sind ϕ_1 und ϕ_2 LTL-Formeln, so sind auch $\phi_1 \wedge \phi_2$ und $\phi_1 \vee \phi_2$ LTL-Formeln.
4. Sind ϕ_1 und ϕ_2 LTL-Formeln, so sind ebenfalls $\mathsf{X}\phi_1$, $\phi_1 \,\mathsf{U}\, \phi_2$ und $\phi_1 \,\mathsf{R}\, \phi_2$ LTL-Formeln.

Die noch verbleibenden aussagenlogischen und temporallogischen Operatoren werden durch die folgenden Abkürzungen definiert:

- $\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
- $\phi_1 \leftrightarrow \phi_2 \equiv (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$
- $\mathsf{F}\,\phi \equiv true \,\mathsf{U}\, \phi$
- $\mathsf{G}\,\phi \equiv false \,\mathsf{R}\, \phi$

Eine LTL-Formel über den Aussagen Π repräsentiert eine Menge unendlicher Wörter über Σ. Um auszudrücken, daß ein Wort $\xi \in \Sigma^\omega$ für eine LTL-Formel ϕ gilt, definieren wir die *Gültigkeitsrelation* $\vDash$. Ist ϕ eine LTL-Formel über Π, so wird $\xi \vDash \phi$ als „ϕ gilt für ξ“ oder auch als „ξ erfüllt ϕ“ gelesen. Wir verwenden des weiteren die Schreibweisen $\xi(0)$ als Bezeichnung des ersten Elementes von ξ und $\xi^{(i)}(x) := \xi(x+i)$ als die Bezeichnung des bei i beginnenden Restwortes von ξ.

Definition 35 (Semantik von LTL)
Sei ξ ein unendliches Wort über Σ. Die *Gültigkeitsrelation* $\vDash$ ist induktiv über der Struktur der LTL-Formeln definiert:

1. Für alle ξ gilt $\xi \vDash true$ und $\xi \nvDash false$.
2. Ist $\pi \in \Pi$, so gilt $\xi \vDash \pi$ gdw. $\pi(\xi(0)) = true$.

3. Ist $\pi \in \Pi$, so gilt $\xi \vDash \neg\pi$ gdw. $\pi(\xi(0)) = false$.
4. $\xi \vDash \phi_1 \wedge \phi_2$ gdw. $\xi \vDash \phi_1$ und $\xi \vDash \phi_2$.
5. $\xi \vDash \phi_1 \vee \phi_2$ gdw. $\xi \vDash \phi_1$ oder $\xi \vDash \phi_2$.
6. $\xi \vDash \mathsf{X}\,\phi$ gdw. $\xi^{(1)} \vDash \phi$.
7. $\xi \vDash \phi_1 \,\mathsf{U}\, \phi_2$ gdw. ein $i \geq 0$ existiert, so daß $\xi^{(i)} \vDash \phi_2$ gilt und für alle $0 \leq j < i$ die Aussage $\xi^{(j)} \vDash \phi_1$ gilt.
8. $\xi \vDash \phi_1 \,\mathsf{R}\, \phi_2$ gdw. für alle $i \geq 0$, für die $\xi^{(i)} \nvDash \phi_2$ gilt, ein $0 \leq j < i$ existiert, so daß $\xi^{(j)} \vDash \phi_1$ gilt.

Wir haben in unserer Definition der LTL-Semantik die Negation nur für die elementaren Aussagen aus Π zugelassen. Diese Beschränkung kann man mit Hilfe der folgenden Beziehungen leicht aufheben.

- $\xi \vDash \neg\neg\phi$ gdw. $\xi \vDash \phi$
- $\xi \vDash \neg\mathsf{X}\,\phi$ gdw. $\xi \vDash \mathsf{X}\,\neg\phi$
- $\xi \vDash \neg(\phi_1 \wedge \phi_2)$ gdw. $\xi \vDash (\neg\phi_1) \vee (\neg\phi_2)$
- $\xi \vDash \neg(\phi_1 \,\mathsf{U}\, \phi_2)$ gdw. $\xi \vDash (\neg\phi_1) \,\mathsf{R}\, (\neg\phi_2)$
- $\xi \vDash \neg(\phi_1 \,\mathsf{R}\, \phi_2)$ gdw. $\xi \vDash (\neg\phi_1) \,\mathsf{U}\, (\neg\phi_2)$

Bei der LTL wird jeder Pfad für sich, also unabhängig von den anderen Pfaden, durch die Formel bewertet. Man definiert daher die Sprache $L(\phi)$ einer LTL-Formel ϕ über Π als die Menge aller unendlichen Wörter über Σ, die ϕ erfüllen: $L(\phi) := \{\, \xi \in \Sigma^\omega \mid \xi \vDash \phi \,\}$.

Satz 18 (Negations-Normalform)
Jede LTL-Formel ϕ läßt sich in eine semantisch äquivalente LTL-Formel ϕ' überführen, bei der die Negation nur auf elementare Aussagen angewandt wird.

Beweis: Durch erschöpfendes Anwenden der obigen syntaktischen Abkürzungen und der semantisch äquivalenten Umformungen. □

Unter der Voraussetzung, daß $|\pi| = |\neg\pi|$ für alle $\pi \in \Pi$ gilt, ändert sich durch die Umformung in Negations-Normalform die Länge der LTL-Formel nicht. Wie wir sehen werden, ist diese Annahme für unseren Verifikationsansatz zulässig.
Im weiteren transformieren wir LTL-Formeln vor einer näheren Betrachtung immer implizit in ihre äquivalente Negations-Normalform.

5.2.2 LTL auf Transitionssystemen

Wir möchten LTL-Formeln benützen, um Eigenschaften der zustandsbasierten Semantik von Transitionssystemen $\mathcal{T} = [Q, \longrightarrow, q_0]$ zu beschreiben. Dazu setzen wir $\Sigma := Q$ und Π als die Menge der elementaren Aussagen auf den Zuständen. Ein Transitionssystem $\mathcal{T}$ erfüllt eine LTL-Formel ϕ, falls $L(\mathcal{T}) \subseteq L(\phi)$, d. h. für alle $\xi \in L(\mathcal{T})$ gilt $\xi \vDash \phi$. Wir erweitern die Gültigkeitsrelation auf Transitionssysteme und schreiben für ein Transitionssystem, das eine LTL-Formel ϕ erfüllt, $\mathcal{T} \vDash \phi$.
Ist das betrachtete Transitionssystem aus dem Erreichbarkeitsgraphen eines sicheren Petrinetzes $\mathcal{N} = [S, T, F, M_0]$ entstanden, so stellen die Zustände des Transitionssystems die Menge der erreichbaren Markierungen des Petrinetzes dar. Die elementaren Aussagen Π auf den Markierungen des Petrinetzes beschränken wir in diesem Fall auf Aussagen π_s mit $s \in S$. Eine elementare Aussage π_s über einer Markierung M ist genau dann wahr, wenn $s \in M$, d. h. s bei M markiert ist. Im weiteren werden wir daher aus Einfachheitsgründen die elementaren Aussagen $\Pi = \{ \pi_s \mid s \in S \}$ in einer LTL-Formel durch die entsprechenden Stellenbezeichner ersetzen.

5.2.3 Beispiele

Zuerst wollen wir uns die allgemeinen Petrinetzeigenschaften Erreichbarkeit von toten Markierungen und Reversibilität von Petrinetzen näher ansehen.

Beispiel 10 (Petrinetzeigenschaften)
Zur Überprüfung, ob ein Petrinetz tote Zustände erreichen kann, müssen wir zuerst die Menge aller möglichen toten Zustände des Petrinetzes beschreiben. Wie wir im Abschnitt 4.2.4 gesehen haben, sind genau die Markierungen tot, die mindestens eine der Vorstellen einer jeden Transition nicht markieren.

$$\mathcal{D}_T := \bigwedge_{t \in T} \bigvee_{s_i \in {}^\bullet t} \neg s_i$$

Ein Petrinetz $\mathcal{N}$ kann also genau dann keine tote Markierung erreichen, wenn alle unendlichen Systemabläufe nur aus Markierungen bestehen, die nicht in $\mathcal{D}$ enthalten sind. Dies läßt sich durch die LTL-Formel $\mathsf{G}\,\neg\mathcal{D}$ beschreiben.
Die Eigenschaft der Reversibilität läßt sich dagegen in der LTL nicht formulieren. Dies wollen wir an dem reversiblen Petrinetz $\mathcal{N}$ in Abbildung 5.1 verdeutlichen.
Nehmen wir einmal an, es gäbe eine systemunabhängige LTL-Formel, mit der wir die Reversibilität von Petrinetzen ausdrücken könnten. Diese Formel müßte dann alle unendlichen Pfade unseres Petrinetzes $\mathcal{N}$ als gültig erkennen, also auch den unendlichen Pfad $\{p_1\} \longrightarrow \{p_2\} \longrightarrow \{p_3\} \longrightarrow \{p_2\} \longrightarrow \cdots$. Entfernen wir nun die gestrichelte Transition aus dem Petrinetz $\mathcal{N}$, so erhalten wir ein Petrinetz, welches nur den genannten unendlichen Pfad besitzt. Da dieser als gültig erkannt wird, würde daraus folgen,

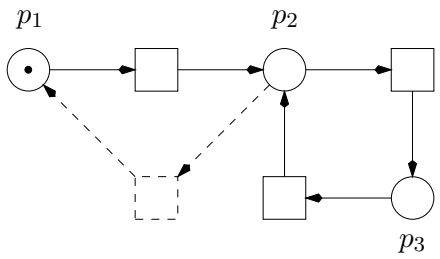


Abbildung 5.1: Ein reversibles Petrinetz.

daß das modifizierte Petrinetz reversibel ist, wodurch ein Widerspruch zur Annahme entsteht.

Beispiel 11 (Produktionsanlage)
In diesem Beispiel wollen wir etwas spezieller werden und eine Produktionsanlage zur Verarbeitung von Metallwerkstücken (vgl. [LL95, HD95, HDS99]) näher betrachten. Die Anlage besteht aus einer Presse zur Verformung der Werkstücke und aus zwei Roboterarmen, die für den An- und Abtransport der Werkstücke zuständig sind. Die Steuerung der Anlage muß für einen reibungslosen Ablauf eine Reihe von Anforderungen erfüllen. Im folgenden beschreiben wir exemplarisch zwei dieser Anforderungen durch LTL-Formeln (vgl. [HD95]).

1. *Sicherheitseigenschaft*: Wenn sich die Presse schließt, befinden sich beide Roboterarme außerhalb ihres Arbeitsbereichs.

$$\begin{aligned}&\mathsf{G}\,(arm1\_loading \vee arm1\_unloading\\ &\quad \rightarrow \neg press\_go\_loadpos \vee \neg press\_go\_unloadpos) \wedge\\ &\mathsf{G}\,(arm2\_loading \vee arm2\_unloading\\ &\quad \rightarrow \neg press\_go\_loadpos \vee \neg press\_go\_unloadpos)\end{aligned}$$

2. *Lebendigkeitseigenschaft*: Befindet sich ein Werkstück innerhalb der Presse, so wird es letztendlich bearbeitet.

$$\mathsf{G}\,(ch\_A1P\_full \rightarrow \mathsf{F}\,blank\_forged)$$

5.3 Büchiautomaten und LTL

Eine LTL-Formel beschreibt, unabhängig vom zu analysierenden System, eine Sprache, die als Menge aller Pfade definiert ist, welche die Formel erfüllen. Ebenso bildet die Menge aller möglichen Pfade durch den Erreichbarkeitsgraphen eines Petrinetzes eine Sprache. Bei den beiden Sprachen handelt es sich jeweils um eine Menge von

gleichstrukturierten unendlichen Wörtern. Jedes dieser unendliche Wörter besteht aus einem endlichen Anfangswort, welches in einen Kreis übergeht. Die beiden Sprachen gehören damit zu der Klasse der sogenannten *ω-regulären Sprachen*.
Ein Petrinetz erfüllt eine LTL-Formel, falls alle Zustandsabläufe die Formel erfüllen, also seine Sprache eine Teilmenge der Formelsprache ist. Das Modelcheckingproblem läßt sich somit auf eine geeignete Darstellungsform der Sprachen und auf einen Inklusionstest reduzieren. Zur Repräsentation von Sprachen eignen sich, wie wir wissen, Automaten. Im Gegensatz zur klassischen Automatentheorie, die sich mit Sprachen, bestehend aus endlichen Wörtern beschäftigt, betrachten wir Automaten für Sprachen mit unendlichen Wörtern. Der Hauptunterschied liegt in der Akzeptierungsbedingung der Wörter. Wir akzeptieren ein unendliches Wort als Eingabe, wenn ein akzeptierender Zustand des Automaten unendlich oft durchlaufen wird. Dieses Kriterium wird *Büchi-Akzeptierungskriterium* genannt und entsprechend werden die Automaten als *Büchiautomaten* bezeichnet.

5.3.1 Büchiautomaten

Definition 36 (Büchiautomat [Tho90])
Ein *(gewöhnlicher) Büchiautomat* ist ein Tupel $\mathcal{A} = [\Sigma, Q, \Delta, q_0, F]$, bestehend aus

1. einem Alphabet Σ,
2. einer endlichen Menge von Zuständen Q,
3. einer Übergangsrelation $\Delta \subseteq Q \times \Sigma \times Q$,
4. einem Anfangszustand q_0 und
5. einer endlichen Menge von akzeptierenden Zuständen $F \subseteq Q$.

Ein Büchiautomat unterscheidet sich strukturell nicht von einem nichtdeterministischen Automaten über endlichen Wörtern. Erst durch das Büchi-Akzeptierungsverhalten wird daraus ein Automat über unendlichen Wörtern.
Ein Lauf σ eines Büchiautomaten für ein unendliches Wort ξ ist eine unendliche Folge $m_0 m_1 m_2 \cdots$ von Zuständen, für die $m_0 = q_0$ und $[m_i, \xi(i), m_{m+1}] \in \Delta$ für alle $i \geq 0$ gilt. Als Grenzverhalten eines Laufes σ betrachten wir die Menge der von σ unendlich oft durchlaufenen Automatenzustände. Wir definieren daher $\lim(\sigma) := \{\, q \in Q \mid \exists^{\omega} i \colon \sigma(i) = q \,\}$. Da die Zustandsmenge Q eines Büchiautomaten endlich ist, ist die Menge $\lim(\sigma)$ nie leer. Darum definieren wir das Akzeptierungsverhalten eines Büchiautomaten folgendermaßen:

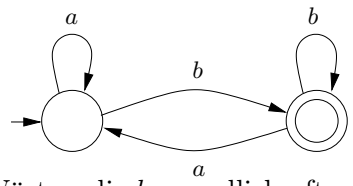
Definition 37 (Büchi-Akzeptierungsverhalten)
Ein Büchiautomat $\mathcal{A} = [\Sigma, Q, \Delta, q_0, F]$ akzeptiert ein unendliches Wort ξ, falls es einen Lauf σ für ξ gibt, mit $\lim(\sigma) \cap F \neq \emptyset$.

Die von einem Büchiautomaten $\mathcal{A}$ akzeptierte Sprache $L(\mathcal{A})$ ist die Menge der unendlichen Wörter, die von $\mathcal{A}$ akzeptiert werden.

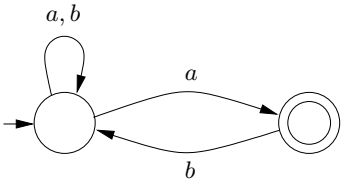
Definition 38 (ω-Regularität)
Eine Sprache L über unendlichen Wörtern heißt *ω-regulär*, falls es einen Büchiautomaten $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

Büchiautomaten beschreiben also gerade die Klasse der ω-regulären Sprachen.

Beispiel 12 (Büchiautomaten ω-regulärer Sprachen)



Wörter, die b unendlich oft enthalten



Unendlich oft ab

Jedes Transitionssystem $\mathcal{T} = [Q, \longrightarrow, q_0]$ läßt sich in einen Büchiautomaten $\mathcal{A}_\mathcal{T} = [Q, Q, \Delta, q_0, Q]$ mit $\Delta := \{\, [q, q, q'] \mid q \longrightarrow q' \,\}$ transformieren, so daß $L(\mathcal{T}) = L(\mathcal{A}_\mathcal{T})$ gilt. Die Sprache $L(\mathcal{T})$ eines Transitionssystems $\mathcal{T}$ ist daher eine ω-reguläre Sprache. Da sich der Erreichbarkeitsgraph $\mathcal{RG}(\mathcal{N}) = [\mathcal{R}_\mathcal{N}(M_0), \mathcal{K}_\mathcal{N}]$ eines Petrinetzes $\mathcal{N} = [S, T, F, M_0]$ wiederum als ein Transitionssystem $\mathcal{T}_\mathcal{N} = [\mathcal{R}_\mathcal{N}(M_0), \mathcal{K}'_\mathcal{N}, M_0]$ interpretieren läßt (vgl. Abschnitt 5.1), existiert ein Büchiautomat $\mathcal{A}_{\mathcal{T}_\mathcal{N}} = [\mathcal{R}_\mathcal{N}(M_0), \mathcal{R}_\mathcal{N}(M_0), \Delta, M_0, \mathcal{R}_\mathcal{N}(M_0)]$ mit $\Delta := \{\, [M, M, M'] \mid [M, M'] \in \mathcal{K}'_\mathcal{N} \,\}$. Es gilt daher $L(\mathcal{N}) = L(\mathcal{A}_{\mathcal{T}_\mathcal{N}})$. Man betrachtet dabei alle erreichbaren Zustände des Petrinetzes als akzeptierende Zustände, so daß alle unendlichen Zustandsabfolgen als Wort der Sprache erkannt werden. Zur einfacheren Beschreibung der Sprachen von LTL-Formeln hat sich eine Erweiterung des Akzeptierungsverhaltens gewöhnlicher Büchiautomaten bewährt.

Definition 39 (Verallgemeinerter Büchiautomat)
Ein *verallgemeinerter Büchiautomat* ist ein Tupel $\mathcal{A} = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$, bestehend aus

1. einem Alphabet Σ,
2. einer endlichen Menge von Zuständen Q,
3. einer Übergangsrelation $\Delta \subseteq Q \times \Sigma \times Q$,
4. einem Anfangszustand q_0 und
5. einer Menge $\mathcal{F} = \{\, F_1, \ldots, F_k \,\}$, welche wiederum aus endlichen Mengen akzeptierender Zustände $F_i \subseteq Q, 1 \leq i \leq k$ besteht.

Verallgemeinerte Büchiautomaten besitzen statt einer Menge akzeptierender Zustände mehrere solcher Mengen. Daher muß auch das Akzeptierungsverhalten angepaßt werden. Dies geschieht naheliegenderweise dadurch, daß man ein unendliches Wort nur dann als Eingabe akzeptiert, wenn es jeweils unendlich oft Zustände jeder der Mengen $F_i \in \mathcal{F}$ durchläuft.

Definition 40 (Verallgemeinertes Büchi-Akzeptierungsverhalten)
Ein verallgemeinerter Büchiautomat $\mathcal{A} = [\Sigma, Q, \Delta, q_0, \mathcal{F}]$ akzeptiert ein unendliches Wort ξ, falls es einen Lauf σ für ξ gibt, mit $\forall F_i \in \mathcal{F}$: $\lim(\sigma) \cap F_i \neq \emptyset$.

Analog zu den gewöhnlichen Büchiautomaten ist die von einem verallgemeinerten Büchiautomaten $\mathcal{A}$ akzeptierte Sprache $L(\mathcal{A})$ die Menge der unendlichen Wörter, die von $\mathcal{A}$ akzeptiert werden.
Jeder verallgemeinerte Büchiautomat läßt sich in einen gewöhnlichen Büchiautomaten transformieren, der die selbe Sprache akzeptiert. Verallgemeinerte Büchiautomaten sind also nicht mächtiger als gewöhnliche Büchiautomaten und können daher auch nur ω-regulären Sprachen repräsentieren.

Satz 19
Jeder verallgemeinerte Büchiautomat $\mathcal{A} = [\Sigma, Q, \Delta, q_0, \{ F_1, \ldots, F_k \}]$ läßt sich in einen gewöhnlichen Büchiautomaten $\mathcal{A}'$ mit $k \cdot |Q|$ Zuständen transformieren, so daß $L(\mathcal{A}) = L(\mathcal{A}')$ gilt.

Beweis: Sei $\mathcal{A} = [\Sigma, Q, \Delta, q_0, \{ F_1, \ldots, F_k \}]$ ein verallgemeinerter Büchiautomat. Dann akzeptiert der daraus konstruierbare gewöhnliche Büchiautomat $\mathcal{A}' = [\Sigma, Q', \Delta', q_0', F']$ mit

- $Q' := Q \times \{ 1, \ldots, k \}$
- $q_0' := (q_0, 1)$
- $\Delta' \subseteq Q' \times \Sigma \times Q'$ ist definiert durch:

$$\begin{aligned} \Delta' \quad := \quad & \{ [(q_1, i), a, (q_2, i)] \mid [q_1, a, q_2] \in \Delta \wedge q_1 \notin F_i \} \cup \\ & \{ [(q_1, i), a, (q_2, (i \bmod k) + 1)] \mid [q_1, a, q_2] \in \Delta \wedge q_1 \in F_i \} \end{aligned}$$

- $F' := F_1 \times \{1\}$

die Sprache $L(\mathcal{A})$.
Diese Konstruktion erstellt k Kopien $\mathcal{A}_1, \ldots, \mathcal{A}_k$ des verallgemeinerten Büchiautomaten $\mathcal{A}$, die dann zu einem gewöhnlichen Büchiautomaten zusammengefügt werden. Man ordnet dazu jedem Teilautomaten jeweils eine der Akzeptierungsmengen $F_1, \ldots, F_k$ zu. Anschließend werden die Zustandsübergänge so modifiziert, daß beim Durchlaufen eines Zustandes aus F_i im Automaten $\mathcal{A}_i$ in den nächst höheren Automaten $\mathcal{A}_{(i \bmod k)+1}$ gewechselt wird.

Erfolgt so ein Wechsel k-mal, so landet man wieder im Ursprungsautomaten. Jede Eingabe, die unendlich oft einen Zustand aus F_1 im Automaten $\mathcal{A}_1$ durchläuft, durchläuft auch, aus Konstruktionsgründen, mindestens einen Zustand aus jeder anderen Akzeptierungsmenge unendlich oft. Somit genügt es, die Akzeptierungsmenge F_1 im Teilautomaten $\mathcal{A}_1$ als alleinige Akzeptierungsmenge des gesamten Automaten zu verwenden. □

Wenn wir also im folgenden Aussagen über Büchiautomaten machen, beziehen wir uns, außer es ist explizit anders angegeben, auf gewöhnliche Büchiautomaten.

Satz 20 (Abgeschlossenheit unter Vereinigung, Schnitt und Komplement)
Seien $\mathcal{A}', \mathcal{A}''$ Büchiautomaten über dem Alphabet Σ. Dann existiert jeweils ein Büchiautomat $\mathcal{A}$ über Σ mit:

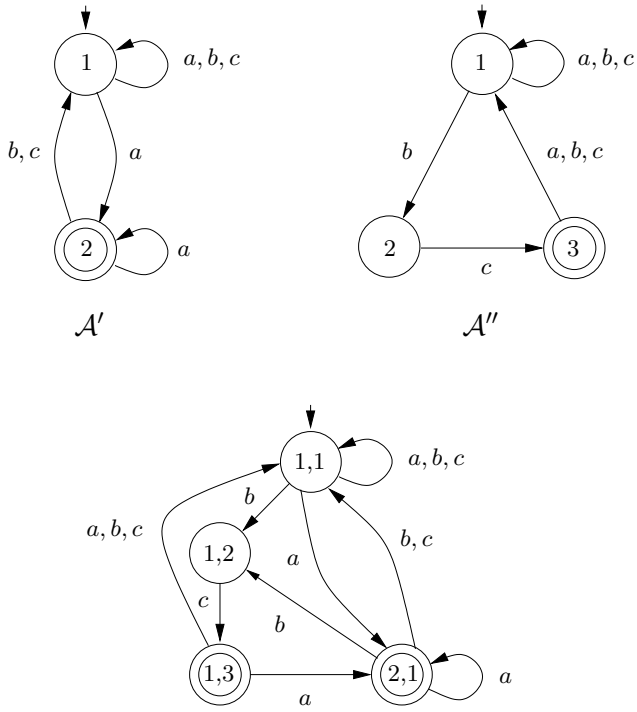
1. $L(\mathcal{A}) = L(\mathcal{A}') \cup L(\mathcal{A}'')$.
2. $L(\mathcal{A}) = L(\mathcal{A}') \cap L(\mathcal{A}'')$.
3. $L(\mathcal{A}) = \overline{L(\mathcal{A}')} = \Sigma^\omega \setminus L(\mathcal{A}')$.

Beweis: Siehe dazu [Büc60] und [Cho74]. □

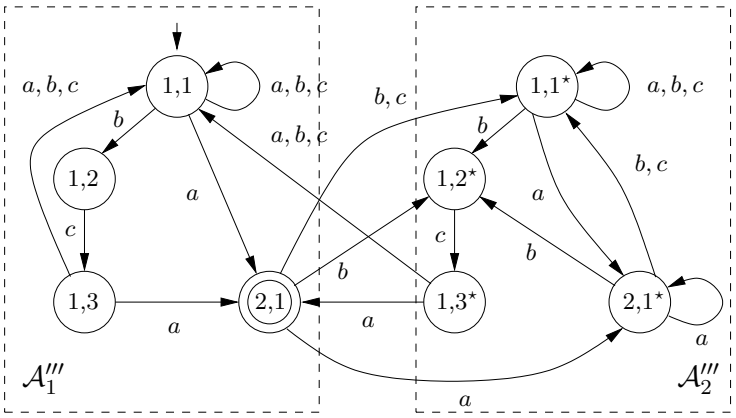
Im allgemeinen ist die Konstruktion eines Büchiautomaten $\mathcal{A}$, der das Komplement einer Sprache $\overline{L(\mathcal{A}')}$ erkennt, exponentiell [Büc60, SVW87]. Wir werden aber später sehen, daß wir das Modelcheckingproblem für LTL-Formeln so umformen können, daß wir nur den Schnitt zweier ω-regulärer Sprachen benötigen. Daher wollen wir dies jetzt näher betrachten.
Seien $\mathcal{A}' = [\Sigma, Q', \Delta', q_0', F']$ und $\mathcal{A}'' = [\Sigma, Q'', \Delta'', q_0'', F'']$ zwei gewöhnliche Büchiautomaten über demselben Alphabet Σ. Ein gewöhnlicher Büchiautomat $\mathcal{A} = [\Sigma, Q, \Delta, q_0, F]$ mit $L(\mathcal{A}) = L(\mathcal{A}') \cap L(\mathcal{A}'')$ läßt sich in zwei Schritten konstruieren. Im ersten Schritt konstruiert man einen verallgemeinerten Büchiautomaten $\mathcal{A}'''$ mit

- $Q''' := Q' \times Q''$
- $q_0''' := (q_0', q_0'')$
- $\Delta''' := \{\, [(q', q''), a, (p', p'')] \mid [q', a, p'] \in \Delta' \wedge [q'', a, p''] \in \Delta'' \,\}$
- $\mathcal{F} := \{\, F' \times Q'', Q' \times F'' \,\}$,

so daß $L(\mathcal{A}''') = L(\mathcal{A}') \cap L(\mathcal{A}'')$ gilt.
Im zweiten Schritt wandelt man den verallgemeinerten Büchiautomaten $\mathcal{A}'''$ mit dem oben angegebenen Verfahren in einen gewöhnlichen Büchiautomaten $\mathcal{A}$ mit $L(\mathcal{A}) = L(\mathcal{A}''')$ um.
Die Abbildung 5.2 illustriert die Konstruktion anhand eines Beispiels.



1. Schritt: Konstruktion eines verallgemeinerten Büchiautomaten $\mathcal{A}'''$



2. Schritt: Konstruktion eines gewöhnlichen Büchiautomaten $\mathcal{A}$

Abbildung 5.2: Schnitt zweier durch die Automaten $\mathcal{A}'$ und $\mathcal{A}''$ repräsentierten Sprachen.

Die Konstruktion des verallgemeinerten Büchiautomaten basiert auf einer synchronen bzw. parallelen Ausführung der Ausgangsautomaten $\mathcal{A}'$ und $\mathcal{A}''$, d. h. einer Produktbildung der beiden Automaten. Dadurch wird garantiert, daß ein unendliches Wort nur dann zu einem Weiterschalten des Automaten $\mathcal{A}'''$ führt, wenn sowohl $\mathcal{A}'$ als auch $\mathcal{A}''$ über diesem Wort schalten können. Ein unendliches Wort wird genau dann akzeptiert, wenn es dazu einen Lauf gibt, der unendlich oft einen Zustand der Akzeptierungsmenge des Automaten $\mathcal{A}'$ sowie auch einen Zustand der Akzeptierungsmenge des Automaten $\mathcal{A}''$ durchläuft. Dies muß allerdings nicht gleichzeitig geschehen. Dadurch unterscheidet sich die Produktbildung zweier Büchiautomaten von der synchronen Produktbildung von Automaten über endlichen Wörtern.

Unser Modelcheckingproblem stellt jedoch einen Spezialfall der obigen Durchschnittskonstruktion dar. Einer der beiden Büchiautomaten wird durch den Erreichbarkeitsgraphen eines Petrinetzes induziert, wodurch seine Akzeptierungsmenge genau der Menge aller seiner Automatenzustände entspricht. Dadurch mutiert eine der bei der Konstruktion des verallgemeinerten Büchiautomaten $\mathcal{A}'''$ auftretenden Akzeptierungsmengen zur Menge aller Zustände von $\mathcal{A}'''$ und kann somit weggelassen werden. Der „verallgemeinerte“ Büchiautomat $\mathcal{A}'''$ ist also schon ein gewöhnlicher Büchiautomat und die Durchschnittskonstruktion entspricht der klassischen synchronen Produktkonstruktion zweier Automaten. Daher gilt für unser Modelcheckingproblem $L(\mathcal{A}' \times \mathcal{A}'') = L(\mathcal{A}') \cap L(\mathcal{A}'')$, was den Gesamtaufwand erheblich reduziert.

5.3.2 Büchiautomaten für LTL-Formeln

Nun wollen wir uns endlich den Kernpunkt der Verifikation, die Konstruktion eines Büchiautomaten aus einer LTL-Formel, näher ansehen. Wir gehen dabei davon aus, daß sich die zu betrachtende LTL-Formel in Negations-Normalform befindet. Die von einer LTL-Formel ϕ definierte Sprache besteht aus unendlichen Wörtern über einem Alphabet Σ. Wir konstruieren daher einen Automaten, der aus einer Menge von Zuständen und beschrifteten Zustandsübergängen besteht. Die Zustände werden mit Teilformeln von ϕ beschriftet. Erreicht man einen Zustand, so besagt seine Beschriftung, welche Teilformeln damit erfüllt sind. Die Beschriftungen der Übergänge ergeben sich daher aus den in einem Zustand vorkommenden Aussagen $\pi_i \in \Pi$ und deren Negationen. Sie bestimmen die Menge der Buchstaben aus unserem Alphabet Σ, die an den Übergängen zu einem Zustand stehen. Eine solche Menge von Aussagen über dem Alphabet Σ repräsentiert dabei genau die Buchstaben aus Σ, die alle Aussagen der Menge erfüllen.

Die Struktur des Automaten ist durch die syntaktische Struktur der Formel bestimmt. Man beginnt eine Büchiautomatenkonstruktion mit einem Startzustand, der mit der LTL-Formel ϕ beschriftet wird. Dieser Zustand wird schrittweise unter Beachtung der

Booleschen Zusammenhänge der Teilformeln und mit Hilfe der Regeln

$$\begin{aligned} \phi_1 \,\mathsf{U}\, \phi_2 &\equiv \phi_2 \vee (\phi_1 \wedge \mathsf{X}\,(\phi_1 \,\mathsf{U}\, \phi_2)) \\ \phi_1 \,\mathsf{R}\, \phi_2 &\equiv \phi_2 \wedge (\phi_1 \vee \mathsf{X}\,(\phi_1 \,\mathsf{R}\, \phi_2)) \\ &\equiv (\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \mathsf{X}\,(\phi_1 \,\mathsf{R}\, \phi_2)) \end{aligned}$$

expandiert. Die Regel für $\phi_1 \,\mathsf{U}\, \phi_2$ besagt z. B., daß man einen so beschrifteten Zustand in zwei Zustände verfeinern kann. Hierbei repräsentiert der eine Zustand die Möglichkeit, daß die Formel ϕ_2 erfüllt ist, und der andere Knoten die Möglichkeit, daß ϕ_1 erfüllt ist und ein Übergang zu einem Zustand existiert, in dem $\phi_1 \,\mathsf{U}\, \phi_2$ erfüllt ist. Der Verfeinerungsprozeß endet, wenn keine neuen Zustände, also Zustände mit noch nicht vorhandenen Beschriftungen, auftreten. Die Akzeptierungsmengen werden durch die in ϕ vorkommenden Until-Teilformeln bestimmt. Für jede Until-Teilformel $\phi_1 \,\mathsf{U}\, \phi_2$ wird eine Akzeptierungsmenge gebildet. Sie enthält alle Zustände, die nicht mit $\phi_1 \,\mathsf{U}\, \phi_2$ beschriftet sind, und alle Zustände, die mit ϕ_2 beschriftet sind. Dadurch erhalten wir einen verallgemeinerten Büchiautomaten.
Wir wollen nun dieses Verfahren näher betrachten. Der im folgenden ausgeführte Algorithmus 12 (LTL2BA) ist eine Modifikation des Verfahrens aus [GPVW95]. Wir verwenden hier aus didaktischen Gründen ein Breitentraversierungsverfahren.
Jeder Zustand des zu konstruierenden Büchiautomaten hat dabei die folgende Struktur:

```
1  BAZustand ≡
2    begin
3      ident : integer;
4      pred : set of integer;
5      new, old, next : set of formula;
6    end.
```

Die zustandslokalen Variablen repräsentieren dabei:

- *ident*: einen eindeutigen Bezeichner des Zustands.
- *pred*: die Menge der Zustände, die einen Übergang zu diesem Zustand besitzen.
- *new*: eine Menge von Formeln, die in diesem Zustand erfüllt sein müssen, aber noch nicht expandiert wurden.
- *old*: eine Menge von Formeln, die in diesem Zustand erfüllt sein müssen und schon expandiert wurden.
- *next*: eine Menge von Formeln, die in allen direkten Nachfolgezuständen dieses Zustandes erfüllt sein müssen.

Wir initialisieren den Algorithmus mit einem besonders gekennzeichneten Startzustand q_0 und einem Nachfolgezustand von q_0, der die Gesamtformel ϕ in *new* enthält.
Zustände werden solange verfeinert, bis keine Formel mehr in *new* vorhanden ist. Diese Verfeinerung erfolgt anhand der syntaktischen Struktur der Formeln in *new*. Als ersten Schritt übertragen wir immer die zu expandierende Formel von *new* nach *old*. Danach unterscheiden wir die folgenden Fälle:

1. $\pi, \neg\pi$ mit $\pi \in \Pi$: Diese Teilformeln werden als atomar betrachtet und daher nicht weiter expandiert. Man überträgt sie nur von *new* nach *old*. Kommt es beim Übertragen zu einer Nichterfüllbarkeit der Formeln in *old*, z. B. durch die Formel $false$ oder durch gleichzeitiges Vorhandensein einer Aussage π und seiner Negation $\neg\pi$, so wird ein solcher Zustand als unerreichbarer Zustand gewertet und entfernt.

2. $\phi_1 \wedge \phi_2$: Die beiden Formeln ϕ_1, ϕ_2 werden in *new* eingefügt.

3. $\phi_1 \vee \phi_2$: Man erzeugt eine Kopie des aktuellen Zustandes. Einem der beiden Zustände wird anschließend die Formel ϕ_1 und dem anderen Zustand die Formel ϕ_2 zu *new* hinzugefügt.

4. $\mathsf{X}\,\phi_1$: Die Formel ϕ_1 kommt nach *next* und wird in einem späteren Schritt weiterbehandelt.

5. $\phi_1 \,\mathsf{U}\, \phi_2$: Man erzeugt eine Kopie des aktuellen Zustandes. Bei einem der beiden Kopien tragen wir ϕ_2 in *new* ein. In der anderen Kopie kommt ϕ_1 nach *new* und $\phi_1 \,\mathsf{U}\, \phi_2$ nach *next*.

6. $\phi_1 \,\mathsf{R}\, \phi_2$: Man erzeugt eine Kopie des aktuellen Zustandes. Bei einem der beiden Zustände tragen wir anschließend ϕ_1 und ϕ_2 in *new* ein. Beim anderen Zustand kommt ϕ_2 nach *new* und $\phi_1 \,\mathsf{R}\, \phi_2$ nach *next*.

Sind alle aktuellen Zustände vollständig verfeinert, befinden sich also keine Formeln mehr in *new*, werden sie in die endgültige Zustandsmenge des Büchiautomaten übertragen. Dabei wird überprüft, ob es für einen neu hinzukommenden Zustand einen äquivalenten, schon vorhandenen Zustand mit der gleichen Beschriftung in *old* und *next* gibt. Ist dies der Fall, so werden die beiden Knoten verschmolzen, d. h. die *pred*-Menge des neuen Zustandes wird zu der des alten Zustandes hinzugefügt. Anschließend wird der neue Zustand gelöscht. Existiert kein äquivalenter Zustand, wird der neue Zustand der Menge der Büchiautomatenzustände hinzugefügt. Für jeden solchen neu hinzukommenden Zustand q wird ein neuer Nachfolgezustand q' erzeugt, wobei die *next*-Formelmenge von q auf die *new*-Menge von q' übertragen wird.
Diese neu erzeugten Zustände besitzen wieder Formeln in *new*, so daß sie auch die ganze Prozedur von Expansion und Einordnung über sich ergehen lassen müssen. Das

ganze Verfahren terminiert, wenn beim Einordnen in die endgültige Büchiautomatenzustandsmenge keine neuen Zustände mehr entstehen.
Dieses Verfahren ist im Algorithmus 12 (LTL2BA) zusammengefaßt. Der Algorithmus verwendet die Funktionen New1, New2 und Next, die wie folgt definiert sind.

ξ	$\phi_1 \wedge \phi_2$	$\phi_1 \,\mathsf{U}\, \phi_2$	$\phi_1 \,\mathsf{R}\, \phi_2$
$\mathsf{New1}(\xi)$	ϕ_1	ϕ_2	$\{\phi_1, \phi_2\}$
$\mathsf{New2}(\xi)$	ϕ_2	ϕ_1	ϕ_2
$\mathsf{Next}(\xi)$	$\emptyset$	$\phi_1 \,\mathsf{U}\, \phi_2$	$\phi_1 \,\mathsf{R}\, \phi_2$

Um aus der so konstruierten Zustandsmenge einen vollständigen Büchiautomaten zu erhalten, benötigen wir noch eine Beschriftung der Übergänge und eine Bestimmung der Mengen von akzeptierenden Zuständen.
Jeder Zustand ist mit den durch ihn erfüllten Teilformeln von ϕ gekennzeichnet. Die Beschriftung der in einen Zustand q mündenden Übergänge ergibt sich daher durch die Buchstaben des Alphabets Σ, die alle in q vorkommenden Aussagen $\pi \in \Pi$ und deren Negationen erfüllen. Man stellt dadurch sicher, daß ein Zustand nur dann erreichbar ist, wenn seine elementaren Aussagen durch das Eingabewort auch erfüllt sind.
In einem ersten Schritt wählen wir die Menge aller Zustände des konstruierten Büchiautomaten als eine Menge von akzeptierenden Zuständen aus. Dadurch akzeptiert unser Büchiautomat zunächst alle unendlichen Wörter, die sich an die Übergangsbeschriftungen halten. Er erfüllt somit fast alle Bedingungen der Semantik einer LTL-Formel. Betrachtet man die Semantik des Until-Operators etwas näher, so fällt auf, daß nur solche Abläufe gültig sind, bei denen ϕ_2 auch wirklich einmal in irgendeinem Zustand erfüllt ist. Die von uns für die Behandlung einer Until-Teilformel verwendete Expansionsregel

$$\phi_1 \,\mathsf{U}\, \phi_2 \quad \equiv \quad \phi_2 \vee (\phi_1 \wedge \mathsf{X}\,(\phi_1 \,\mathsf{U}\, \phi_2))$$

erlaubt aber auch Zustandsabfolgen, in denen unendlich oft ϕ_1 vorkommt, ohne daß ein Zustand erreicht wird, in dem ϕ_2 gültig ist. Wir müssen also, um die korrekte Semantik des Until-Operators zu modellieren, solche ungültigen Zustandsabfolgen ausschließen. Dazu ersetzen wir die zuerst definierte Akzeptierungsmenge durch die sich aus den Until-Teilformeln ergebenden Akzeptierungsmengen. Die Expansionsregel garantiert, daß eine in einem Zustand vorkommende Until-Teilformel $\phi_1 \,\mathsf{U}_i\, \phi_2$ solange vorkommt, bis ein mit ϕ_2 beschrifteter Zustand erreicht wird. Daher genügt es zu überprüfen, ob ein Zustandsablauf existiert, der unendlich oft durch einen Zustand geht, in dem $\phi_1 \,\mathsf{U}_i\, \phi_2$ nicht vorkommt, oder durch einen Zustand geht, der mit ϕ_2 beschriftet ist. Daraus ergibt sich für die in der zu verifizierenden LTL-Formel ϕ vorkommenden Until-Teilformeln $\phi_1 \,\mathsf{U}_1\, \phi_2, \ldots, \phi_1 \,\mathsf{U}_m\, \phi_2$ die verallgemeinerte Büchi-Akzeptierungsbedingung $\mathcal{F} := \{\, \Phi_1, \ldots, \Phi_m \,\}$ mit $\Phi_i := \{q \in Q \mid \phi_2 \in q.old \vee (\phi_1 \,\mathsf{U}_i\, \phi_2) \notin q.old\,\}$.

Algorithmus 12 (Erzeugung eines Büchiautomaten aus einer LTL-Formel)

```
1  LTL2BA(φ) ≡
2    FinalStates := {q0};
3
4    proc putToFinal(expanded) ≡
5      unexpanded := ∅;
6      forall q ∈ expanded do
7        if ∃ q' ∈ FinalStates : (q.old = q'.old ∧ q.next = q'.next) then
8          q'.pred := q'.pred ∪ q.pred
9        else
10         FinalStates := FinalStates ∪ {q};
11         q'' := newState();
12         q''.pred := {q};q''.new := q.next;
13         unexpanded := unexpanded ∪ {q''}
14       fi
15     od;
16     if unexpanded ≠ ∅ then expand(unexpanded) fi
17   end putToFinal;
18
19   proc expand(unexpanded) ≡
20     expanded := ∅;
21     while unexpanded ≠ ∅ do
22       q := oneof(unexpanded); unexpanded := unexpanded \ {q};
23       while q.new ≠ ∅ do
24         ξ := oneof(q.new); q.new := q.new \ {ξ};
25         q.old := q.old ∪ {ξ};
26         case ξ of
27           ξ = π or ξ = ¬π:
28             if ξ = false ∨ ¬ξ ∈ q.old then unexpanded := unexpanded \ {q} fi;
29           ξ = X φ1:
30             q.next := q.next ∪ {φ1};
31           ξ = φ1 ∧ φ2:
32             q.new := q.new ∪ ({φ1, φ2} \ q.old);
33           ξ = φ1 ∨ φ2 or ξ = φ1 U φ2 or ξ = φ1 R φ2:
34             q' := dupNode(q);
35             q.new := q.new ∪ (New1(ξ) \ q.old);
36             q'.new := q'.new ∪ (New2(ξ) \ q'.old);
37             q'.next := q'.next ∪ Next(ξ);
38             unexpanded := unexpanded ∪ {q'}
39         esac
40       od;
41       expanded := expanded ∪ {q}
42     od
43     putToFinal(expanded)
44   end expand;
45
46 begin
47   q1 := newState();
48   q1.pred := {q0}; q1.new := {φ};
49   expand({q1});
50   return FinalStates
51 end.
```

Die Übergangsrelation Δ des so konstruierten Büchiautomaten ist nicht total. Benötigt das verwendete Verifikationsverfahren diese Eigenschaft, muß der Büchiautomat noch vervollständigt werden. Dazu führt man einen weiteren speziellen Zustand *error* ein, der einen mit allen Buchstaben des Alphabets Σ beschrifteten Übergang zu sich selber besitzt. Anschließend wird von jedem Zustand des Büchiautomaten ein Übergang zum Zustand *error* erzeugt. Diese Übergänge beschriftet man jeweils mit den Buchstaben aus Σ, die nicht in den Beschriftungen der vom betrachteten Zustand ausgehenden Übergängen enthalten sind. Die Zustandsübergangsrelation wird dadurch total und alle ungültigen Abläufe werden in den nichtakzeptierenden Zustand *error* umgeleitet. Das vorgestellte Konstruktionsverfahren ist auf beliebige LTL-Formeln anwendbar. Die maximale Größe des dabei konstruierten Büchiautomaten ist durch die Anzahl der unterschiedlichen LTL-Teilformeln bestimmt. Es gilt daher:

Satz 21 (Größe des Büchiautomaten einer LTL-Formel)
Zu einer gegebenen LTL-Formel ϕ läßt sich ein gewöhnlicher Büchiautomat $\mathcal{A} = [\Sigma, Q, \Delta, q_0, F]$ mit $|Q|$ in $2^{\mathcal{O}(|\phi|)}$ konstruieren, so daß $L(\mathcal{A}) = L(\phi)$ gilt.

Beweis: Siehe [VW94, Var96]. □

Aus der Definition ω-regulärer Sprachen folgt damit die folgende Aussage:

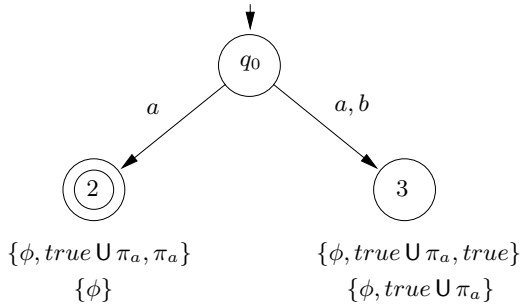
Korollar 3
Die durch eine LTL-Formel definierte Sprache ist ω-regulär.

5.3.3 Beispiel

Betrachten wir die LTL-Formel $\phi = \mathsf{GF}\pi_a$ über dem Alphabet $\Sigma = \{a, b\}$. Die Aussage $\pi_a \in \Pi$ ist definiert durch $\pi_a(a) = 1$ und $\pi_a(b) = 0$. Zur Konstruktion eines Büchiautomaten transformieren wir die Formel ϕ zuerst in Negations-Normalform $\phi = false\,\mathsf{R}\,(true\,\mathsf{U}\,\pi_a)$. Als nächstes protokollieren wir die einzelnen Durchläufe durch den Algorithmus. Dazu geben wir jeweils die Menge der noch nicht verfeinerten, der verfeinerten und der endgültigen Büchiautomatenzustände an. Mit q_0 bezeichnen wir den beim Verfahren implizit eingeführten Startzustand.

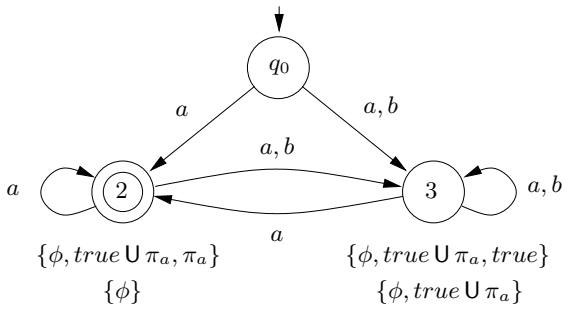
1. Durchlauf:

ident	*new*	*old*	*next*	*pred*
1	$\{\phi\}$	$\emptyset$	$\emptyset$	q_0
2	$\emptyset$	$\{\phi, true\,\mathsf{U}\,\pi_a, \pi_a\}$	$\{\phi\}$	q_0
3	$\emptyset$	$\{\phi, true\,\mathsf{U}\,\pi_a, true\}$	$\{\phi, true\,\mathsf{U}\,\pi_a\}$	q_0



2. Durchlauf:

ident	*new*	*old*	*next*	*pred*
4	$\{\phi\}$	$\emptyset$	$\emptyset$	2
5	$\{\phi, true \mathsf{U} \pi_a\}$	$\emptyset$	$\emptyset$	3
6	$\emptyset$	$\{\phi, true \mathsf{U} \pi_a, \pi_a\}$	$\{\phi\}$	2
7	$\emptyset$	$\{\phi, true \mathsf{U} \pi_a, true\}$	$\{\phi, true \mathsf{U} \pi_a\}$	2
8	$\emptyset$	$\{\phi, true \mathsf{U} \pi_a, \pi_a\}$	$\{\phi\}$	3
9	$\emptyset$	$\{\phi, true \mathsf{U} \pi_a, true\}$	$\{\phi, true \mathsf{U} \pi_a\}$	3



Wir erhalten für unsere Formel $\mathsf{GF}\pi_a$ den Büchiautomaten $\mathcal{A}_{\mathsf{GF}\pi_a} = [\Sigma, Q, \Delta, q_0, F]$ mit

- $\Sigma := \{a, b\}$,
- $Q := \{q_0, 2, 3\}$,
- $\Delta := \{[q_0, \{a\}, 2], [q_0, \{a, b\}, 3], [2, \{a\}, 2], [2, \{a, b\}, 3], [3, \{a, b\}, 3], [3, \{a\}, 2]\}$,
- $q_0 := q_0$ und
- $F := \{2\}$.

5.4 Modelchecking

Bei dem von uns betrachteten Modelcheckingproblem handelt es sich um die Frage „Erfüllt ein nebenläufiges System $\mathcal{N}$ die LTL-Spezifikation ϕ?“.
Wie wir gesehen haben, lassen sich sowohl das System als auch die LTL-Formel durch einen Büchiautomaten repräsentieren, so daß sich unser Modelcheckingproblem auch als das automatentheoretische Problem

$$\text{„}L(\mathcal{N}) \subseteq L(\phi)?\text{“}$$

beschreiben läßt. Dies kann man noch weiter zu

$$\begin{aligned} L(\mathcal{N}) \subseteq L(\phi) &\equiv L(\mathcal{N}) \cap \overline{L(\phi)} = \emptyset \\ &\equiv L(\mathcal{N}) \cap L(\neg\phi) = \emptyset \end{aligned}$$

umformen (vgl. [Var96]). Dadurch eröffnet sich uns ein einfaches Modelcheckingverfahren.
Einen Büchiautomaten für unser nebenläufiges System $\mathcal{N}$ können wir über den Erreichbarkeitsgraph erzeugen und ebenso einen Büchiautomaten für eine beliebige LTL-Formel, also auch für $\neg\phi$. Den Schnitt zweier durch Büchiautomaten beschriebener Sprachen erhalten wir durch die Produktbildung der beiden Automaten. Zur Entscheidung unseres Modelcheckingproblems fehlt uns also nur noch ein Verfahren, mit dem wir feststellen können, ob eine durch einen Büchiautomaten beschriebene Sprache leer ist.
Nehmen wir einmal an, die Sprache $L(\mathcal{A})$ sei nicht leer. Dann existiert ein unendliches Wort $\xi \in \Sigma^\omega$ und damit ein Lauf σ des Büchiautomaten $\mathcal{A}$, d. h. eine unendliche Folge $m_0 m_1 m_2 \cdots$ von Zuständen, für die $m_0 = q_0$ und $[m_i, \xi(i), m_{i+1}] \in \Delta$ für alle $i \geq 0$ gilt. Dieser Lauf σ durchläuft unendlich oft einen Zustand aus der Menge der akzeptierenden Zustände des Automaten. Betrachtet man den Büchiautomaten als einen Graphen, so existiert damit ein unendlicher Weg durch den Graphen, der zwangsläufig in einem Kreis endet, der einen akzeptierenden Zustand enthält.
Die Überprüfung, ob eine Sprache $L(\mathcal{A})$ leer ist, läßt sich somit durch einen einfachen Tiefendurchlauf des Graphen bewerkstelligen (vgl. Algorithmus 13 (Tiefensuche)). Erreicht man bei der Tiefensuche einen Zustand, der auf dem aktuellen Suchweg schon einmal vorgekommen ist, überprüft man, ob dazwischen ein akzeptierender Zustand liegt. Findet man einen solchen Kreis, kann man die weitere Suche abbrechen, da die Sprache mindestens diesen Ablauf enthält, also nicht leer ist.
Die in solch einem Fall auf dem durch die Tiefensuche angelegten Stapel liegenden Zustände beschreiben einen Ablaufpfad des Systems, der die Spezifikation nicht erfüllt, also ein Gegenbeispiel. Durch Simulation dieses Ablaufes kann man daher das Verhalten des Systems beobachten, das zu einer Verletzung der Spezifikationseigenschaften führt, und gegebenenfalls Veränderungen vornehmen.

Algorithmus 13 (Tiefensuche)

```
1  Tiefensuche([Σ, Q, Δ, q0, F]) ≡
2    VisitedStates := ∅;
3    Stack := ∅;
4    count := 0;
5
6    proc DFS(q) ≡
7      VisitedStates := VisitedStates ∪ {q};
8      count := count + 1;
9      q.num := count;
10     Stack := Stack ∪ {q};
11     forall { q' | [q, _, q'] ∈ Δ } do
12       if q' ∈ Stack then
13         if { q'' | q'' ∈ Stack ∧ q''.num ≥ q'.num } ∩ F ≠ ∅ then
14           exit("accepting cycle found")
15         fi
16       elsif q' ∉ VisitedStates then
17         DFS(q')
18       fi
19     od;
20     Stack := Stack \ {q};
21   end DFS;
22
23 begin
24   DFS(q0);
25   exit("no accepting cycle found")
26 end.
```

Der automatentheoretische Ansatz liefert uns ein einfaches Modelcheckingverfahren, bei dem das Leerheitsproblem für Büchiautomaten, wie wir gesehen haben, auf ein Erreichbarkeitsproblem für Graphen reduziert wird. Die Überprüfung, ob vom Initialzustand aus ein akzeptierender Zustand erreichbar ist, der auf einem Kreis liegt, ist in linearer Zeit bezüglich der Graphgröße möglich [Tar72]. Auf die dabei verwendete Zerlegung des Graphen in maximale stark-zusammenhängende Komponenten gehen wir näher im Abschnitt 5.5 ein.
Es ergibt sich daraus die folgende Komplexitätsaussage.

Satz 22
Das Leerheitsproblem für durch Büchiautomaten repräsentierte Sprachen ist in linearer Zeit entscheidbar.

Beweis: Siehe [EL85a, EL85b]. □

Die Negation einer LTL-Formel und die Überführung in Negations-Normalform verändert die Länge der Formel nicht. Aus Satz 21 wissen wir, daß die Größe eines Büchiautomaten einer LTL-Formel ϕ in $2^{\mathcal{O}(|\phi|)}$ liegt. Des weiteren ergibt sich die Größe des

Produkts zweier Büchiautomaten als das Produkt der Einzelgrößen, woraus sich mit dem obigen Satz die folgenden Komplexitätsaussagen für das Modelcheckingproblem ergeben.

Satz 23 (Komplexität für das LTL-Modelchecking)

1. *Die Komplexität des Modelcheckingproblems ist PSPACE-vollständig.*
2. *Die Überprüfung, ob das nebenläufige System $\mathcal{N}$ die LTL-Formel ϕ erfüllt, ist mit einem Zeitaufwand von $\mathcal{O}(|\mathcal{T}_{\mathcal{N}}| \cdot 2^{\mathcal{O}(|\phi|)})$ entscheidbar.*

Beweis: Siehe [LP85, SC85, VW86]. □

Eine polynomielle obere Zeitschranke bzgl. der Systemgröße und eine exponentielle obere Zeitschranke bzgl. der Größe der Spezifikation erscheinen akzeptabel, da die Spezifikationen normalerweise ziemlich klein sind [LP85].

5.5 Verbesserungen

Die Überprüfung eines nebenläufigen Systems $\mathcal{N}$ anhand einer LTL-Spezifikation ϕ läßt sich in zusammenfassend in sieben Schritte unterteilen.

1. Negation der Formel ϕ.
2. Überführung von $\neg\phi$ in Negations-Normalform.
3. Konstruktion eines verallgemeinerten Büchiautomaten $\mathcal{A}_{\neg\phi}$.
4. Konstruktion eines gewöhnlichen Büchiautomaten $\mathcal{A}'_{\neg\phi}$
5. Erzeugung des gewöhnlichen Büchiautomaten $\mathcal{A}_{\mathcal{T}_{\mathcal{N}}}$ aus dem Erreichbarkeitsgraphen des Systems.
6. Bildung des Produktautomaten $\mathcal{A}'_{\neg\phi} \times \mathcal{A}_{\mathcal{T}_{\mathcal{N}}}$.
7. Überprüfung von $L(\mathcal{A}_{\mathcal{T}_{\mathcal{N}}}) = \emptyset$.

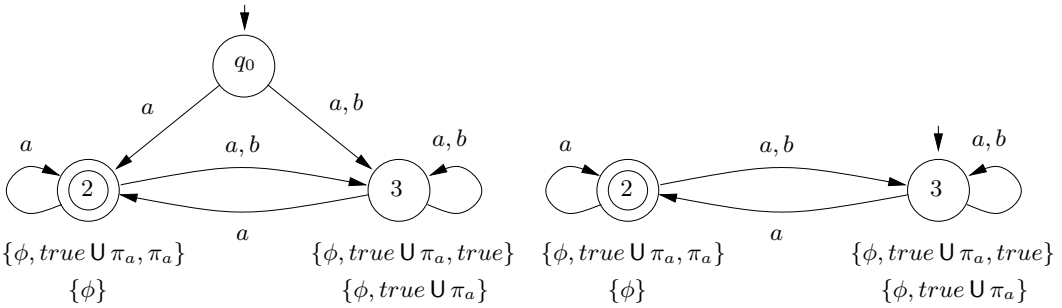
Einige dieser Schritte bieten Optimierungsmöglichkeiten, die den Aufwand des Modelcheckings in vielen Fällen stark reduzieren können.

Die Umformung der Formel $\neg\phi$ in Negations-Normalform kann durch semantikerhaltende Transformationen ergänzt werden. Sie dienen dazu, die Komplexität der Formel, also die Länge, zu reduzieren. In der Tabelle 5.1 sind einige der möglichen Transformationen aufgeführt.

$\neg true$	$\equiv$	$false$	$\mathsf{X}\, true$	$\equiv$	$true$
$\neg false$	$\equiv$	$true$	$\mathsf{X}\, false$	$\equiv$	$false$
$\phi \wedge \phi$	$\equiv$	ϕ	$\phi \vee \phi$	$\equiv$	ϕ
$false \wedge \phi$	$\equiv$	$false$	$false \vee \phi$	$\equiv$	ϕ
$\phi \wedge false$	$\equiv$	$false$	$\phi \vee false$	$\equiv$	ϕ
$true \wedge \phi$	$\equiv$	ϕ	$true \vee \phi$	$\equiv$	$true$
$\phi \wedge true$	$\equiv$	ϕ	$\phi \vee true$	$\equiv$	$true$
$\neg\phi \wedge \phi$	$\equiv$	$false$	$\neg\phi \vee \phi$	$\equiv$	$true$
$\phi \wedge \neg\phi$	$\equiv$	$false$	$\phi \vee \neg\phi$	$\equiv$	$true$
$\mathsf{X}\,\phi_1 \wedge \mathsf{X}\,\phi_2$	$\equiv$	$\mathsf{X}\,(\phi_1 \wedge \phi_2)$	$\mathsf{X}\,\phi_1 \vee \mathsf{X}\,\phi_2$	$\equiv$	$\mathsf{X}\,(\phi_1 \vee \phi_2)$
$\phi\,\mathsf{U}\,\phi$	$\equiv$	ϕ	$\phi\,\mathsf{R}\,\phi$	$\equiv$	ϕ
$\phi\,\mathsf{U}\, false$	$\equiv$	$false$	$\phi\,\mathsf{R}\, true$	$\equiv$	$true$
$false\,\mathsf{U}\,\phi$	$\equiv$	ϕ	$true\,\mathsf{R}\,\phi$	$\equiv$	ϕ
$\phi\,\mathsf{U}\, true$	$\equiv$	$true$	$\phi\,\mathsf{R}\, false$	$\equiv$	$false$
$\neg\phi\,\mathsf{U}\,\phi$	$\equiv$	$true\,\mathsf{U}\,\phi$	$\neg\phi\,\mathsf{R}\,\phi$	$\equiv$	$false\,\mathsf{R}\,\phi$
$\phi\,\mathsf{U}\,\neg\phi$	$\equiv$	$true\,\mathsf{U}\,\neg\phi$	$\phi\,\mathsf{R}\,\neg\phi$	$\equiv$	$false\,\mathsf{R}\,\neg\phi$
$(\mathsf{X}\,\phi_1)\,\mathsf{U}\,(\mathsf{X}\,\phi_2)$	$\equiv$	$\mathsf{X}\,(\phi_1\,\mathsf{U}\,\phi_2)$	$(\mathsf{X}\,\phi_1)\,\mathsf{R}\,(\mathsf{X}\,\phi_2)$	$\equiv$	$\mathsf{X}\,(\phi_1\,\mathsf{R}\,\phi_2)$
$true\,\mathsf{U}\,(\mathsf{X}\,\phi)$	$\equiv$	$\mathsf{X}(true\,\mathsf{U}\,\phi)$	$false\,\mathsf{R}\,(\mathsf{X}\,\phi)$	$\equiv$	$\mathsf{X}(false\,\mathsf{R}\,\phi)$
$\phi_1\,\mathsf{U}\,(true\,\mathsf{U}\,\phi_2)$	$\equiv$	$true\,\mathsf{U}\,\phi_2$	$\phi_1\,\mathsf{R}\,(false\,\mathsf{R}\,\phi_2)$	$\equiv$	$false\,\mathsf{R}\,\phi_2$
$true\,\mathsf{U}\,(\phi_1\,\mathsf{U}\,\phi_2)$	$\equiv$	$true\,\mathsf{U}\,\phi_2$	$false\,\mathsf{R}\,(\phi_1\,\mathsf{R}\,\phi_2)$	$\equiv$	$false\,\mathsf{R}\,\phi_2$
$(\phi_1\,\mathsf{U}\,\phi_2) \vee (\phi_1\,\mathsf{U}\,\phi_3)$	$\equiv$	$\phi_1\,\mathsf{U}\,(\phi_2 \vee \phi_3)$	$(\phi_1\,\mathsf{R}\,\phi_2) \wedge (\phi_1\,\mathsf{R}\,\phi_3)$	$\equiv$	$\phi_1\,\mathsf{R}\,(\phi_2 \wedge \phi_3)$
$(\phi_1\,\mathsf{U}\,\phi_3) \wedge (\phi_2\,\mathsf{U}\,\phi_3)$	$\equiv$	$(\phi_1 \wedge \phi_2)\,\mathsf{U}\,\phi_3$	$(\phi_1\,\mathsf{R}\,\phi_3) \vee (\phi_2\,\mathsf{R}\,\phi_3)$	$\equiv$	$(\phi_1 \vee \phi_2)\,\mathsf{R}\,\phi_3$
$\mathsf{X}\,(true\,\mathsf{U}\,(false\,\mathsf{R}\,\phi))$	$\equiv$	$true\,\mathsf{U}\,(false\,\mathsf{R}\,\phi)$	$\mathsf{X}\,(false\,\mathsf{R}\,(true\,\mathsf{U}\,\phi))$	$\equiv$	$false\,\mathsf{R}\,(true\,\mathsf{U}\,\phi)$
$true\,\mathsf{U}\,(false\,\mathsf{R}\,(true\,\mathsf{U}\,\phi))$	$\equiv$	$false\,\mathsf{R}\,(true\,\mathsf{U}\,\phi)$			
$false\,\mathsf{R}\,(true\,\mathsf{U}\,(false\,\mathsf{R}\,\phi))$	$\equiv$	$true\,\mathsf{U}\,(false\,\mathsf{R}\,\phi)$			
$false\,\mathsf{R}\,(true\,\mathsf{U}\,\phi_1) \vee false\,\mathsf{R}\,(true\,\mathsf{U}\,\phi_2)$	$\equiv$	$false\,\mathsf{R}\,(true\,\mathsf{U}\,(\phi_1 \vee \phi_2))$			

Tabelle 5.1: Vereinfachung von LTL-Formeln durch Ersetzen mit semantisch äquivalenten Formeln.

Das Verfahren zur Konstruktion eines Büchiautomaten für eine LTL-Formel [GPVW95], auf das wir uns im Abschnitt 5.3.2 beziehen, war ursprünglich so konzepiert worden, daß es die Konstruktion eines Büchiautomaten während des Verifikationsprozesses erlaubt.
Wir erzeugen mit unserem Algorithmus 12 (LTL2BA) aber den kompletten Büchiautomaten vor der eigentlichen Verifikation. Dies bietet uns die Möglichkeit, den Büchiautomaten durch Verschmelzen von äquivalenten Zuständen zu reduzieren. Als äquivalente Zustände betrachten wir Zustände, die Ausgangskanten zu den selben Knoten haben, wobei alle entweder zur Menge der akzeptierenden Zustande oder alle zur Menge der nicht akzeptierenden Zustände gehören.
Der im Abschnitt 5.3.3 konstruierte Büchiautomat $\mathcal{A}_{\mathsf{GF}\pi_a}$ hat drei Zustände und läßt sich durch Zustandsverschmelzung zu einem Automaten mit zwei Zuständen reduzieren.



Die Tabelle 5.2 vergleicht, anhand einiger exemplarischer LTL-Formeln, das Büchiautomatenkonstruktionsverfahren aus [GPVW95], in der Implementierung aus Abschnitt 5.3.2 (LTL2BA), mit dem durch die hier vorgestellten Vereinfachungsregeln ergänzten Verfahren (AdvLTL2BA). Die in der Tabelle mit „+“ gekennzeichneten Verfahren beziehen sich auf eine in [GPVW95] vorgestellte leichte Optimierung des ursprünglichen Büchiautomatenkonstruktionsverfahrens.
Wie aus der Tabelle ersichtlich ist, ergeben sich durch die zusätzlichen Vereinfachungsregeln z. T. erhebliche Reduktionen der Büchiautomatengröße.
Der Erreichbarkeitsgraph eines Petrinetzes kann sehr groß werden. Erzeugt man daraus, zusammen mit dem Büchiautomaten einer LTL-Formel, den Produktautomaten, so nimmt der Speicherplatzbedarf nochmals drastisch zu.
Eine Möglichkeit, mit diesem Problem umzugehen, stellt die *on-the-fly* Erzeugung des Produktautomaten dar. Man erzeugt dabei den Produktautomaten nur so weit, wie er aktuell im Verifikationsprozeß benötigt wird. Ein Vorteil gegenüber der klassischen vollständigen Erzeugung des Produktautomaten ergibt sich dann, wenn seine Sprache nicht leer ist. In diesem Fall können wir nach dem Erkennen eines akzeptierenden Laufes abbrechen und kommen damit unter Umständen mit der Konstruktion eines kleinen Teils des Produktautomaten aus.

Formeln	LTL2BA	LTL2BA$^+$	AdvLTL2BA	AdvLTL2BA$^+$
$\mathsf{FG}\,(\phi_1 \wedge \mathsf{F}\,\phi_2)$	6	6	4	4
$\mathsf{FG}\,\phi_1 \wedge \mathsf{GF}\,\phi_2$	11	7	5	4
$(\phi_1\,\mathsf{U}\,\phi_1) \vee (\phi_2\,\mathsf{U}\,\phi_1)$	10	6	3	2
$\mathsf{G}(\phi_1 \rightarrow \mathsf{F}\,\phi_2)$	6	6	3	3
$\neg\mathsf{G}\,(\phi_1 \rightarrow \mathsf{X}\,(\phi_2\,\mathsf{R}\,\phi_3))$	6	6	5	5
$\mathsf{GF}\,\phi_1 \rightarrow \mathsf{GF}\,\phi_2$	10	6	6	5
$\mathsf{G}\,((\phi_1 \wedge \phi_2) \rightarrow \phi_1\,\mathsf{U}\,(\phi_3 \wedge \phi_4))$	8	8	3	3
$\mathsf{G}((\phi_1 \vee \phi_2) \rightarrow (\phi_3 \wedge \phi_4))$	3	3	1	1
$\mathsf{GF}\,\phi_1 \vee \mathsf{GF}\,\phi_2$	9	5	5	5
$\mathsf{X}\,(\mathsf{GF}\,(\mathsf{X}\,\phi_1 \wedge \mathsf{X}\,\phi_2))$	6	6	4	4
$\phi_1\,\mathsf{R}\,(\mathsf{G}\,(\phi_2\,\mathsf{U}\,(\mathsf{F}\,\phi_3)))$	16	10	2	2
Gesamt:	91	69	41	38

Tabelle 5.2: Vergleich verschiedener Konstruktionsverfahren von Büchiautomaten.

Modifiziert man die zur Verifikation verwendete Tiefensuche leicht, kann man auch für den Fall, daß die Sprache leer ist, Einsparungen des Speicherplatzbedarfs erreichen. Das abgewandelte Verfahren basiert auf einer Zerlegung des Produktautomaten in *maximale stark-zusammenhängende Komponenten* (kurz SZKs). Unter einer maximalen stark-zusammenhängenden Komponente eines Graphen versteht man eine maximale Menge von Knoten eines gerichteten Graphen, wobei von jedem Knoten dieser Menge aus alle anderen Knoten der Menge erreichbar sind. Diese Zerlegung des Produktautomaten in SZKs kann man in linearer Zeit bezüglich der Automatengröße bewerkstelligen [Tar72] (vgl. Algorithmus 14 (SZK)).

Existiert eine vom Initialzustand aus erreichbare SZK des Produktautomaten, die einen akzeptierenden Zustand enthält, so existiert trivialerweise ein Kreis von diesem Zustand zu sich selber. Damit ist die Sprache des Produktautomaten nicht leer.

Die SZKs eines Graphen bilden selber einen gerichteten azyklischen Graphen.

Bei der on-the-fly Erzeugung des Produktautomaten kann man, um Speicherplatz zu sparen, eine gefundene SZK aus dem Speicher entfernen, da alle von dieser SZK erreichbaren SZKs schon früher betrachtet wurden. Durch das Entfernen von Zuständen besteht nun aber die Möglichkeit, daß Zustände mehrmals duchlaufen werden. Die lineare Zeitkomplexität wird in diesem Fall für eine Speicherplatzeinsparung aufgegeben.

Es besteht also die Möglichkeit, daß nicht der ganze Produktautomat im Speicher gehalten werden muß, auch wenn man ihn im Verlauf des Verfahrens schrittweise erzeugt.

Reaktive Systeme sind normalerweise so konzipiert, daß sie nicht terminieren. Daher kommt es häufig vor, daß alle erreichbaren Zustände des Systems sich gegenseitig erreichen können, also in einer SZK liegen. Für solche Systeme ergibt sich also meist keine Speicherplatzersparnis.

Algorithmus 14 (SZK)

```
1  SZK([Σ, Q, Δ, q0, F]) ≡
2    VisitedStates := ∅;
3    Stack := ∅;
4    count := 0;
5
6    proc visit(q) ≡
7      VisitedStates := VisitedStates ∪ {q};
8      count := count + 1;
9      q.num := count;
10     q.uplink := count;
11     Stack := Stack ∪ {q};
12     forall { q' | [q, _, q'] ∈ Δ } do
13       if q' ∉ VisitedStates then
14         visit(q');
15         q.uplink := min(q.uplink, q'.uplink)
16       elsif q' ∈ Stack then
17         q.uplink := min(q.uplink, q'.num)
18       fi
19     od;
20     if q.num = q.uplink then
21       szk := { q' | q' ∈ Stack ∧ q'.num ≥ q.num };
22       if szk ∩ F ≠ ∅ then
23         exit("accepting SZK found")
24       fi;
25       Stack := Stack \ szk
26     fi
27   end visit;
28
29 begin
30   visit(q0);
31   exit("no accepting SZK found")
32 end.
```

Zusätzlich zu dem Speicherplatz der Zustände wird bei der Berechnung der SZKs auch noch Speicherplatz für Verwaltungsinformationen $(uplink, num)$ benötigt.
Ein Verfahren, welches mit nur zwei extra Bits pro Zustand auskommt, ist die *verschachtelte Tiefensuche* (s. Algorithmus 15 (nDFS)). Erreicht man bei der ersten Tiefensuche einen akzeptierenden Zustand, so startet man von diesem Zustand aus eine zweite Tiefensuche, um zu überprüfen, ob ein Kreis zu diesem Zustand existiert. Dabei protokolliert man jeden erreichten Zustand separat für die beiden Tiefendurchläufe. Dadurch wird im schlimmsten Fall der Produktautomat zweimal durchlaufen [CVWY91, GH93, HP96]. Die Speichereffizienz diese Verfahrens kann dadurch erhöht werden, daß man zur Protokollierung der schon besuchten Zustände eine Hashtabelle ohne Kollisionsauflösung verwendet, die nur ein Bit pro Zustand benötigt. Dadurch akzeptiert man einen Fehler des Analyseverfahrens, der aber durch geschickte Wahl der Hashfunktion sehr klein gehalten werden kann [Hol88].

Die vorgestellten Verfahren zur Überprüfung der Leerheit einer Sprache lassen sich auch auf verallgemeinerte Büchiautomaten erweitern. Man muß dabei überprüfen, ob es einen Kreis gibt, der aus jeder akzeptierenden Zustandsmenge $F_i \in \mathcal{F}$ einen Zustand enthält. Die Transformation des verallgemeinerten Formelbüchiautomaten in einen gewöhnlichen Büchiautomaten wird damit überflüssig.

5.6 Zusammenfassende Bemerkungen

Die von uns hier betrachteten reaktive Systeme stellen eine wichtige Gruppe von nebenläufigen Systemen dar. Sie zeichnen sich dadurch aus, daß sich ihr Verhalten mit endlich vielen Zuständen adäquat beschreiben läßt. Wir verwenden daher zur Modellierung solcher Systeme Stellen/Transitions-Netze. Deren zustandsbasierte Interleaving-Semantik wird durch den Erreichbarkeitsgraphen repräsentiert. Dieser läßt sich auf einfache Weise in ein äquivalentes Transitionssystem beziehungsweise in einen Büchiautomaten überführen.
Zur Spezifikation von Systemeigenschaften verwenden wir die LTL. Man unterscheidet traditionell zwei Klassen von LTL-Modelcheckingalgorithmen. Zu einer Klasse gehören die Verfahren, die direkt auf der Struktur der Formel arbeiten [LP85, BCG95a]. Die zweite Klasse konstruiert in einem Zwischenschritt einen Büchiautomaten aus der Formel, so daß der Modelcheckingprozeß auf einem automatentheoretischen Verfahren beruht [VW86, CVWY91]. Die Büchiautomatendarstellung ist oft um einiges kompakter als die Ausgangsformel, wodurch sich der zusätzliche Umformungsaufwand meist lohnt.
LTL-Formeln spezifizieren Eigenschaften, die für alle Abläufe eines Systems gelten müssen. Dies entspricht einem All-Quantor über der Pfadmenge. Eine interessante Be-

Algorithmus 15 (Verschachtelte Tiefensuche)

```
1  nDFS([Σ, Q, Δ, q0, F]) ≡
2    VisitedStates := ∅;
3    seed := q0;
4
5    proc 2DFS(q) ≡
6      VisitedStates := VisitedStates ∪ {(q, 1)};
7      forall { q' | [q, _, q'] ∈ Δ } do
8        if (q', 1) ∉ VisitedStates then
9          2DFS(q')
10       elsif q = seed then
11         exit("accepting cycle found")
12       fi
13     od;
14   end 2DFS;
15
16   proc DFS(q) ≡
17     VisitedStates := VisitedStates ∪ {(q, 0)};
18     forall { q' | [q, _, q'] ∈ Δ } do
19       if (q', 0) ∉ VisitedStates then DFS(q') fi
20     od;
21     if q ∈ F then
22       seed := q;
23       2DFS(q)
24     fi
25   end DFS;
26
27 begin
28   DFS(q0);
29   exit("no cycle found")
30 end.
```

obachtung ist nun, daß man das vorgestellte Modelcheckingverfahren auch zur Überprüfung von Eigenschaften verwenden kann, die nur die Existenz von mindestens einem erfüllenden Ablauf fordern. Dies entspricht wiederum einem Existenz-Quantor über der Pfadmenge. Die einzige dafür erforderliche Veränderung des Verfahrens besteht darin, die Negation der Formel zu unterlassen. Existiert ein erfüllender Pfad, so erfüllt auch das System in diesem Fall die Spezifikation.
Die Komplexität des LTL-Modelcheckingverfahrens ist exponentiell bezüglich der Länge der LTL-Formel. Da die Spezifikationen im Vergleich zur Größe des Erreichbarkeitsgraphen des zu überprüfenden Systems [LP85] meist sehr klein sind, ist der lineare Aufwand bezüglich der Produktautomatengröße die für das LTL-Modelcheckingverfahren ausschlaggebende Komplexitätsangabe.
Dem großen Speicherplatzbedarf des Produktautomaten versucht man durch eine on-the-fly Konstruktion zu begegnen. Dabei geht man von dem Formelautomaten und dem Erreichbarkeitsgraphen aus und erzeugt den Produktautomaten schrittweise während der Verifikation. Da aber schon der Erreichbarkeitsgraph sehr groß werden kann, wäre ein Verfahren von Vorteil, welches den Produktautomaten während der Verifikation direkt aus dem Petrinetz erzeugt. Dadurch läßt sich möglicherweise verhindern, daß der komplette Erreichbarkeitsgraph erzeugt wird. Einem solchen Verfahren wollen wir uns im nächsten Kapitel widmen.

6 LTL-Modelchecking mit binären Entscheidungsgraphen

Das im vorherigen Kapitel vorgestellte automatentheoretische Verfahren zum LTL-Modelchecking basiert auf der Suche eines Gegenbeispiels zur gegebenen LTL-Spezifikation. Ist ein solches Gegenbeispiel gefunden, so kann der Verifikationsprozeß abgebrochen werden. Benötigt das Verfahren zur Erkennung eines Gegenbeispiels nur einen Teil des Produktbüchiautomaten, kann man dieses mit einer on-the-fly Konstruktion des Zustandsraumes koppeln, so daß auch nur der notwendige Teilproduktbüchiautomat im Speicher gehalten werden muß.

Traditionell wird der Produktbüchiautomat on-the-fly aus dem Transitionssystem des modellierten Systems und dem Formelbüchiautomaten erzeugt.

In unserem Fall entspricht das Transitionssystem dem Erreichbarkeitsgraphen des Petrinetzmodells. Dieser kann an sich schon sehr groß werden und wird möglicherweise gar nicht vollständig zur Konstruktion eines Gegenbeispiels benötigt. Weiterhin besteht durch den Koppelungsmechanismus des Erreichbarkeitsgraphen mit dem Formelbüchiautomaten die Möglichkeit, daß bestimmte Zustände des Erreichbarkeitsgraphen im Kontext der zu widerlegenden Formel nicht erreichbar sind. Konstruiert man also vorab den Erreichbarkeitsgraphen, führt dies möglicherweise zu einem unnötigen Zeitaufwand und Speicherplatzbedarf.

Aus diesen Gründen suchen wir eine Möglichkeit, den Produktbüchiautomaten direkt aus dem Systempetrinetz und dem Formelbüchiautomaten ohne Umwege zu konstruieren. Dazu übertragen wir den Büchiautomatenformalismus auf Petrinetze, was uns zu den sogenannten „*Büchinetzen*“ führt. Sie ermöglichen uns eine Koppelung des Systempetrinetzes und des Formelbüchiautomaten auf Petrinetzebene, so daß der sich daraus ergebende Erreichbarkeitsgraph äquivalent zum ursprünglichen Produktbüchiautomaten ist.

Darauf aufbauen lassen sich die on-the-fly Verifikationsverfahren aus dem Kapitel 5. Sie basieren auf Tiefensucheverfahren, die mit linearem zeitlichen Aufwand bzgl. der Produktautomatengröße arbeiten. Der on-the-fly Ansatz erweist sich vor allem während des Systementwicklungsprozesses als sehr hilfreich, da dort häufig Fehler auftreten, die so schnell wie möglich erkannt werden müssen. Ist am Ende des Entwicklungsprozesses eine abschließende Überprüfung aller vorgegebenen Spezifikationen notwendig,

wenn man also davon ausgeht, daß das System die Spezifikationen erfüllt, sind meist symbolische Verfahren vorteilhafter. Dies belegt auch die Studie [DDR+99].
Das von E. M. Clarke, O. Grumberg und K. Hamaguchi entworfene symbolische LTL-Modelcheckingverfahren [CGH94] basiert auf einer Breitentraversierung des Produktbüchiautomaten, welche sich nicht für eine on-the-fly Verifikation eignet. Des weiteren benötigen sie einen quadratischen Aufwand bzgl. der erreichbaren Zustände des Produktbüchiautomaten.
Diese Gegenüberstellung wirft nun die Frage nach einem Verfahren auf, welches die Vorteile beider Verfahren, d. h. on-the-fly Verifikation mit linearem zeitlichen Aufwand und kompakte Darstellung großer Zustandsräume, vereinigt.

6.1 Büchinetze

In der Arbeit von J. Esparza und S. Melzer [EM97] wird der Büchiautomatenformalismus auf die Ebene der Petrinetze gehoben. Dazu wird eine spezielle Klasse von Petrinetzen, die sogenannten *Büchinetze*, eingeführt, deren Erreichbarkeitsgraphen Büchiautomaten widerspiegeln.

Definition 41 (Büchinetze)
Ein *Büchinetz* ist ein Tupel $\mathcal{B} = [\mathcal{N}, S_F]$ bestehend aus einem sicheren Stellen/Transitions-Netz $\mathcal{N} = [S, T, F, M_0]$ und einer Menge von akzeptierenden Stellen $S_F \subseteq S$.

Als Erreichbarkeitsgraphen $\mathcal{RG}(\mathcal{B})$ eines Büchinetzes $\mathcal{B}$ definieren wir den Erreichbarkeitsgraphen des zugeordneten Petrinetzes $\mathcal{N}$, dessen Übergangsrelation im Falle von toten Markierungen um Schleifen (der toten Markierungen zu sich selber) ergänzt wird. Die Menge aller von M_0 aus erreichbaren Markierungen $\mathcal{R}_{\mathcal{B}}(M_0)$ des Büchinetzes $\mathcal{B}$ entspricht der Menge aller erreichbaren Markierungen $\mathcal{R}_{\mathcal{N}}(M_0)$ des zugrundeliegenden Petrinetzes $\mathcal{N}$. Da es sich bei den Büchinetzen um eine Übertragung des Büchiautomatenformalismus handelt, benötigen wir noch die Definition des Akzeptierungsverhaltens eines Büchinetzes.

Definition 42 (Akzeptierungsverhalten eines Büchinetzes)
Ein Lauf eines Büchinetzes $\mathcal{B} = [\mathcal{N}, S_F]$ ist eine unendliche Folge $M_0 M_1 M_2 \cdots$ von Zuständen aus $\mathcal{R}_{\mathcal{B}}(M_0)$ mit $[M_i, \_, M_{i+1}] \in \mathcal{K}_{\mathcal{B}}$, $i \in \mathbb{N}$. Ein Lauf σ wird von einem Büchinetz genau dann akzeptiert, wenn eine Markierung aus $\mathcal{F} := \{ M_i \mid M_i \cap S_F \neq \emptyset \}$ existiert, die von σ unendlich oft durchlaufen wird.

Als Sprache $L(\mathcal{B})$ eines Büchinetzes $\mathcal{B}$ bezeichnen wir die Menge aller von $\mathcal{B}$ akzeptierten Läufe. Wie bei den Büchiautomaten stellt sich die Frage nach der Komplexität des Leerheitsproblems für durch Büchinetze repräsentierte Sprachen.

Satz 24

1. *Das Leerheitsproblem für durch Büchinetze repräsentierte Sprachen ist PSPACE-vollständig.*

2. *Die Überprüfung, ob die Sprache eines Büchinetzes $\mathcal{B} = [[S, T, F, M_0], S_F]$ leer ist, ist mit einem Zeitaufwand von $\mathcal{O}(|\mathcal{RG}(\mathcal{B})|$ entscheidbar.*

Beweis: Zu 1. siehe [EM97].
Zu 2.: Das Leerheitsproblem kann, wie wir wissen, auf ein Erreichbarkeitsproblem im Erreichbarkeitsgraphen reduziert werden, welches in linearer Zeit bezüglich der Graphgröße entscheidbar ist (s. Abschnitt 5.4). □

Jedes Petrinetz $\mathcal{N} = [S, T, F, M_0]$ läßt sich als ein Büchinetz $\mathcal{B}_\mathcal{N} = [\mathcal{N}, S]$ auffassen, so daß die Sprache des Petrinetzes $L(\mathcal{N})$ gleich der Sprache des Systembüchinetzes $L(\mathcal{B}_\mathcal{N})$ ist.
Auch jeder Büchiautomat $\mathcal{A} = [\Sigma, Q, \Delta, q_0, F_\mathcal{A}]$ läßt sich in ein entsprechendes Büchinetz $\mathcal{B}_\mathcal{A} = [[S, T, F, M_0], S_F]$ mit

1. $S := Q$,

2. $T := \Delta$,

3. $F := \{ (q', \delta), (\delta, q'') \mid \exists\, \delta \in \Delta \colon \delta = [q', \_, q''] \}$,

4. $M_0 := \{q_0\}$ und

5. $S_F := F_\mathcal{A}$

transformieren. Die Sprache des Büchinetzes $\mathcal{B}_\mathcal{A}$ entspricht dabei der Menge der akzeptierten Läufe des Büchiautomaten $\mathcal{A}$.
Durch entsprechende Koppelung des Systempetrinetzes mit dem als Petrinetz dargestellten Formelbüchiautomaten, läßt sich ein Produktbüchinetz erzeugen, dessen Erreichbarkeitsgraph dem Produktbüchiautomaten des entsprechenden Modelcheckingproblems entspricht.
Bei den von uns verwendeten sicheren Stellen/Transitions-Netzen kann man das Schalten einer Transition nicht von der Unmarkiertheit einer Stelle abhängig machen. Daher benötigen wir für die von uns angestrebte Koppelung des Systempetrinetzes mit dem Formelbüchinetz eine explizite Darstellung der Negation elementarer Aussagen.
Wir fügen deshalb für jede Stelle $s \in S$, die als negierte Teilformel in der betrachteten LTL-Formel ϕ vorkommt, eine sogenannte komplementäre Stelle $\overline{s}$ dem Systempetrinetz hinzu.
Seien $s, \overline{s} \in S$. Eine Stelle $\overline{s}$ ist eine komplementäre Stelle zu s, falls $\overline{s}\bullet = \bullet s \setminus s\bullet$, $\bullet\overline{s} = s\bullet \setminus \bullet s$ und $\overline{s} \in M_0 \iff s \notin M_0$ gilt. Eine Stelle und ihre komplementäre Stelle

bilden eine 1-S-Invariante, so daß unter jeder erreichbaren Markierung $M \in \mathcal{R}_{\mathcal{N}}(M_0)$ immer genau eine der beiden Stellen markiert ist.
Die LTL-Formel ϕ passen wir auch entsprechend an, indem wir die in ϕ vorkommenden Teilformeln der Form $\neg s$ durch die zu s komplementäre Stelle $\overline{s}$ ersetzen.
Durch das Ergänzen um komplementäre Stellen verändert sich das Schaltverhalten des Petrinetzes nicht. Daher bleibt die Sprache des Petrinetzes, wenn man die Markierungen auf die ursprünglichen Stellen einschränkt, ebenfalls erhalten. Analog gilt dies für die modifizierte LTL-Formel.
Wir betrachten daher im weiteren nur noch auf diese Art angepaßte Systempetrinetze und LTL-Spezifikationen.
Da unsere angestrebte Koppelung auf Büchinetzebene arbeiten soll, also direkt auf den Stellen des Systempetrinetzes, verändern wir noch die Beschriftung der Übergänge des Formelbüchiautomaten. Anstatt den an den Übergängen stehenden Markierungen verwenden wir die Menge der sie repräsentierenden elementaren Aussagen. Wir ersetzen sie also durch die Menge der Stellen, die durch die im Endzustand eines Überganges vorkommenden elementaren Teilformeln $s \in \Pi$ bestimmt sind. Damit erhalten wir einen Formelbüchiautomaten über dem Alphabet 2^S.
Die Idee der Koppelung des Petrinetzes $\mathcal{N}$ mit dem in ein Büchinetz transformierten Formelbüchiautomaten $\mathcal{A}_{\neg\phi}$ basiert auf der Betrachtung des Formelbüchinetzes als ein an $\mathcal{N}$ angekoppelten Beobachter. Das Formelbüchinetz überwacht sozusagen die Schaltvorgänge des Systempetrinetzes und ermöglicht dadurch das Erkennen akzeptierender Abläufe.
Das Produktbüchinetz muß dabei so konstruiert werden, daß es dem Schaltverhalten des Produktbüchiautomaten $\mathcal{A}_{\mathcal{T}_{\mathcal{N}}} \times \mathcal{A}_{\neg\phi}$ entspricht. Von einer Markierung (M_1, q_1) kann also genau dann zu einer Markierung (M_2, q_2) gewechselt werden, wenn

1. $\mathcal{N}$ von M_1 nach M_2 wechseln kann,
2. ein Übergang $[q_1, \{s_0, \ldots, s_n\}, q_2] \in \Delta_{\neg\phi}$ des Formelbüchiautomaten $\mathcal{A}_{\neg\phi}$ existiert und
3. $\{s_0, \ldots, s_n\} \subseteq M_1$

gilt.
Wir simulieren das angestrebte Schaltverhalten durch alternierendes Schalten des Formelbüchinetzes und des Systempetrinetzes. Das Formelbüchinetz überprüft, ob die aktuelle Markierung M_1 des Systempetrinetzes alle Stellen aus $\{s_0, \ldots, s_n\}$ markiert. Ist dies der Fall, wechselt das Formelbüchinetz vom Zustand q_1 nach q_2 und übergibt die Aufforderung zum Schalten an das Systempetrinetz. Dieses schaltet eine schaltfähige Transition und gibt die Kontrolle zurück an das Formelbüchinetz. Zur Steuerung dieses alternierenden Schaltverhaltens führen wir die beiden Stellen $sc_{\mathcal{N}}$ und $sc_{\neg\phi}$ ein, wobei eine Marke auf $sc_{\mathcal{N}}$ ($sc_{\neg\phi}$) bedeutet, daß das Systempetrinetz (das Formelbüchinetz) als nächstes schalten darf.

Definition 43 (Produktbüchinetz)
Ein Produktbüchinetz $\mathcal{B}_\times = [[S_\times, T_\times, F_\times, M_{0_\times}], S_{F_\times}]$ eines Petrinetzes $\mathcal{N} = [S, T, F, M_0]$ und eines Formelbüchiautomaten $\mathcal{A}_{\neg\phi} = [2^S, Q, \Delta, q_0, F_{\neg\phi}]$ ist definiert durch:

1. $S_\times := S \cup Q \cup \{sc_\mathcal{N}, sc_{\neg\phi}\}$
2. $T_\times := T \cup \Delta$
3. $F_\times := F \cup$
   $\{ (q', \delta), (\delta, q'') \mid \forall\, \delta \in \Delta \colon \delta = [q', \_, q''] \} \cup$
   $\{ (sc_\mathcal{N}, t), (t, sc_{\neg\phi}) \mid \forall t \in T \} \cup$
   $\{ (sc_{\neg\phi}, \delta), (\delta, sc_\mathcal{N}) \mid \forall \delta \in \Delta \} \cup$
   $\{ (s_0, \delta), (\delta, s_0), \ldots, (s_n, \delta), (\delta, s_n) \mid \forall \delta \in \Delta \colon \delta = [q', \{s_0, \ldots, s_n\}, q''] \}$
4. $M_{0_\times} := M_0 \cup \{q_0, sc_{\neg\phi}\}$
5. $F_\times := F_{\neg\phi}$.

Entfernt man aus dem durch das Produktbüchinetz definierten Erreichbarkeitsgraphen alle Zustände, in denen $sc_\mathcal{N}$ markiert ist, und beschränkt anschließend die Markierungen auf die Stellen des Petrinetzes und des Formelautomaten, d. h. man entfernt $sc_\mathcal{N}$ und $sc_{\neg\phi}$, so erhält man einen zum Produktbüchiautomaten $\mathcal{A}_{\mathcal{T}_\mathcal{N}} \times \mathcal{A}_{\neg\phi}$ äquivalenten Graphen.

Satz 25
Sei $\mathcal{N}$ ein sicheres Stellen/Transitions-Netz und sei $\mathcal{A}_{\neg\phi}$ der zu der Negation der LTL-Formel ϕ gehörige Büchiautomat. Für das sich daraus ergebende sichere Produktbüchinetz $\mathcal{B}_\times$ gilt: $\mathcal{T}_\mathcal{N} \vDash \phi$ gdw. $L(\mathcal{B}_\times) = \emptyset$.

Beweis: Siehe [EM97]. □

Das Produktbüchinetz ist offensichtlich sicher, da das Systempetrinetz sicher ist und die Stellen $sc_\mathcal{N}, sc_{\neg\phi}$ sowie die Stellen von $\mathcal{A}_{\neg\phi}$ jeweils eine 1-S-Invariante bilden.
Das Beispiel in Abbildung 6.1 stellt ein Produktbüchinetz dar, bestehend aus einem Systempetrinetz $\mathcal{N}$, welches um eine komplementäre Stelle $\overline{p}$ ergänzt wurde, und einem Formelbüchiautomaten $\mathcal{B}_{\mathsf{GF}\,\overline{p}}$, der die Negation der LTL-Formel $\mathsf{FG}\,p$ repräsentiert.
Unser Modelcheckingproblem haben wir durch die Produktbüchinetzkonstruktion in ein Erreichbarkeitsproblem auf Erreichbarkeitsgraphen von Büchinetzen transformiert. Dies erlaubt es uns, den Erreichbarkeitsgraphen immer nur soweit zu erzeugen, wie es für den aktuellen Stand der Verifikation notwendig ist. Die Produktbüchinetze bieten uns also die Möglichkeit eines on-the-fly Verifikationsverfahrens.
Zur Erzeugung und zur Handhabung großer Zustandsmengen (d. h. großer Erreichbarkeitsgraphen) haben wir im Kapitel 4 binäre Entscheidungsgraphen eingeführt. Im folgenden wollen wir nun aufzeigen, wie die symbolische Behandlung von Zustandsräumen zur Lösung unseres Modelcheckingproblems verwendet werden kann.

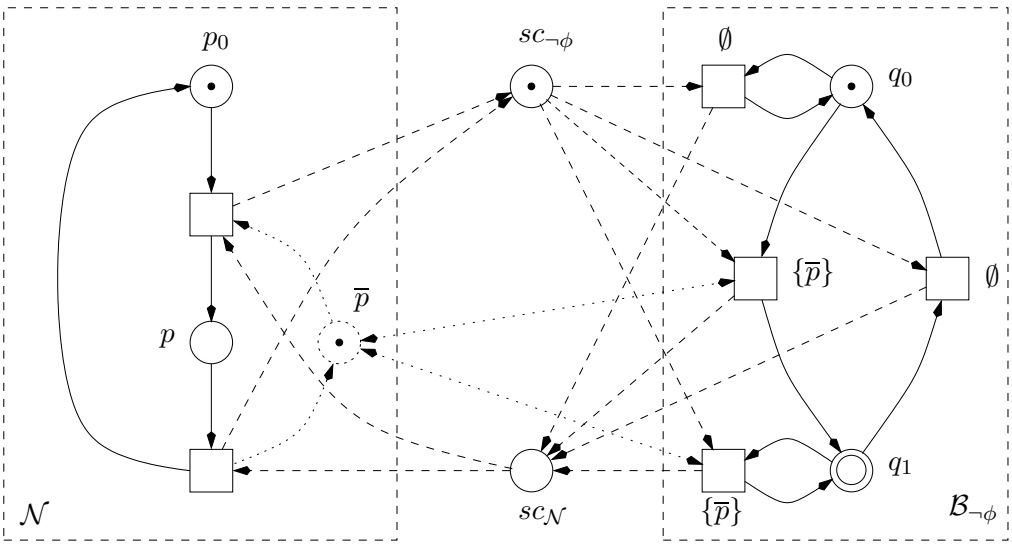


Abbildung 6.1: Produktbüchinetz bestehend aus einem Petrinetz $\mathcal{N}$ und einem Formelbüchiautomaten $\mathcal{B}_{\mathsf{GF}\,\overline{p}}$.

6.2 Symbolische Verifikation von LTL

Zur symbolischen Berechnung aller unendlichen Pfade im Erreichbarkeitsgraphen eines Büchinetzes benötigen wir Funktionen, die zu einer Menge von Zuständen die Menge der direkten Vor- bzw. Nachfolgezustände ermitteln. Wir greifen dazu auf die Zustandsübergangsfunktionen pre_T und $post_T$ für Petrinetze aus dem Abschnitt 4.2 zurück.
Da die Übergangsrelation eines Büchinetzerreichbarkeitsgraphen per Definition total ist, müssen wir die beiden Funktionen noch entsprechend anpassen. Wir wollen dies hier exemplarisch für die Nachfolgerfunktion $post_T$ durchführen. Die Anpassungen für die Vorgängerfunktion pre_T sind analog.
Um der Übergangssemantik eines Büchinetzerreichbarkeitsgraphen gerecht zu werden, müssen wir bei der Bestimmung der direkten Nachfolger einer Zustandsmenge $\mathcal{M}$ alle in $\mathcal{M}$ vorkommenden toten Zustände der Nachfolgerzustandsmenge hinzufügen. Die Menge der toten Markierungen eines Petrinetzes sind dabei durch

$$\mathcal{D}_T := \bigwedge_{t \in T} \bigvee_{s_i \in {}^\bullet t} \neg s_i$$

gegeben (vgl. Kapitel 4.2.4). Damit definieren wir die Nachfolgerfunktion *next* für Büchinetze als:

$$next\colon 2^{(2^S)} \longrightarrow 2^{(2^S)}, \quad next(\mathcal{M}) := post_T(\mathcal{M}) \cup (\mathcal{D}_T \cap \mathcal{M}).$$

Eine wichtige Eigenschaft für die Handhabbarkeit von binären Entscheidungsgraphen ist deren Größe, die, wie wir wissen, stark von der Anzahl der zur Codierung der Zustände benötigten Variablen abhängt. Unsere Produktbüchinetzkonstruktion vergrößert die benötigte Anzahl von Stellen durch das Hinzufügen von komplementären Stellen und von Stellen zur Koordinierung des Wechsels der beiden Teilnetze. Dies führt bei der Codierung der Zustände zu einer entsprechenden Erhöhung der Variablenanzahl.
Erweitert man die klassischen Stellen/Transitions-Netze um sogenannte *Inhibitorkanten*, erübrigt sich beim Konstruktionsprozeß eines Produktbüchiautomaten das Hinzufügen von komplementären Stellen. Inhibitorkanten sind spezielle Kanten von Stellen zu Transitionen, die das Schaltverhalten der Transitionen beeinflussen. Eine Transition, die durch Inhibitorkanten verbundene Vorstellen besitzt, kann nur dann schalten, wenn diese Stellen unmarkiert sind. Damit ist es möglich, das Schalten von Transitionen von der Unmarkiertheit bestimmter Stellen abhängig zu machen, wodurch wir leicht die Negiertheit von elementaren Aussagen der LTL-Formel ausdrücken können. Die symbolische Beschreibung des Schaltverhaltens eines Stellen/Transitions-Netzes (vgl. Abschnitt 4.2) läßt sich auf einfache Weise auf Netze mit Inhibitorkanten erweitern [PRCB94].
Die zur Koordinierung des wechselseitigen Schaltens des Systempetrinetzes und des Formelbüchinetzes eingeführten Stellen $sc_{\mathcal{N}}$ und $sc_{\neg\phi}$ lassen sich dadurch einsparen, daß man die Unterteilung des Schaltens eines Produktbüchinetzes direkt in der Nachfolgerfunktion *next* codiert. Zuerst werden also alle Nachfolgemarkierungen bzgl. des Formelbüchinetzes erzeugt, von denen dann die Nachfolger bzgl. des Systempetrinetzes generiert werden. Die Funktion next stellt eine Implementierung dieses Ansatzes dar.

Algorithmus 16 (next)

```
1  func next(M) ≡
2     /* 1. Schritt : Schalten des Formelbüchinetzes */
3     New1 := post_{T_¬φ}(M);
4     /* 2. Schritt : Schalten des Systempetrinetzes */
5     Dead := D_{T_N} ∩ New1;
6     New2 := post_{T_N}(New1 \ Dead);
7     return New2 ∪ Dead
8  end next;
```

Es genügt dabei, nur tote Markierungen bzgl. des Systempetrinetzes zu betrachten. Haben wir einen vervollständigten Formelbüchiautomaten, kann dieser bei jeder Markierung schalten. Ist der Formelbüchiautomat nicht vervollständigt, werden durch für ihn tote Markierungen solche Pfade beschrieben, die sich nicht an die Übergangsbeschreibung des Formelbüchiautomaten halten. Es handelt sich also um ungültige Pfade, die uns sowieso nicht interessieren und die daher auch nicht weiter verfolgt werden müssen.

Eine weitere Einsparung der Variablenanzahl ergibt sich ohne zusätzlichen Aufwand aus der Tatsache, daß die Stellen eines Formelbüchiautomaten eine 1-S-Invariante bilden. Wir können also einfach die Stellenmenge eines Formelbüchiautomaten binär codieren (vgl. Abschnitt 4.2.5) und benötigen dafür nur eine logarithmische Anzahl von Variablen.
Mit den angegebenen Modifikationen ist die benötigte Variablenanzahl des Produktbüchiansatzes gleich der Variablenanzahl, die wir auch für die Codierung des entsprechenden Produktbüchiautomaten benötigen würden. Der Übergang zum Formalismus der Produktbüchinetze erfordert also keinen zusätzlichen Mehraufwand an Speicherplatz.
Im weiteren bezeichnen wir die zur Nachfolgerfunktion *next* analoge Vorgängerfunktion einer Büchiautomatenzustandsmenge mit *prev* und die dazugehörige Implementierung mit prev.
Zur Berechnung aller erreichbaren Zustände eines Büchinetzes $\mathcal{R}_{\mathcal{B}}(M_0)$ können wir auf den Algorithmus 7 (ReachableSet) aus dem Abschnitt 4.2.3 zurückgreifen.
Der letzte noch verbleibende Schritt zur Lösung unseres Modelcheckingproblems besteht nun darin, einen Kreis im Erreichbarkeitsgraphen des Produktbüchinetzes zu finden, der einen akzeptierenden Zustand enthält.
Für dieses Problem wird klassisch das Verfahren von E. A. Emerson und C.-L. Lei aus dem Jahre 1986 [EL86] verwendet. Es basiert auf zwei ineinander geschachtelten Fixpunktiterationen. Der Algorithmus 17 (BwdCycleCheck) ist eine für unsere Bedürfnisse angepaßte Variante dieses Verfahrens.
Die Menge $\mathcal{F}$ bezeichnet dabei die Menge aller akzeptierenden Markierungen des Büchinetzes.
Die innere Fixpunktiteration berechnet alle Vorgänger der aktuellen Markierungsmenge $\mathcal{Y}$. Sie startet mit der aktuellen Markierungsmenge $\mathcal{Y}$ und erzeugt, bis zum Erreichen eines Fixpunktes, eine aufsteigende Inklusionskette von Markierungsmengen. Bezeichne $\mathcal{Z}_i$ den i-ten Iterationsschritt, dann gilt für ein $k \in \mathbb{N}$:

$$\mathcal{Y} = \mathcal{Z}_0 \subset \mathcal{Z}_1 \subset \cdots \subset \mathcal{Z}_k = \mathcal{Z}_{k+1}.$$

Die äußere Fixpunktiteration beginnt mit der Menge aller akzeptierenden Markierungen $\mathcal{F}$ des Büchinetzes. Sie stellt den Startwert für die Menge $\mathcal{Y}$ dar. Bei jeder Iteration wird unter Zuhilfenahme der inneren Fixpunktiteration die Menge aller akzeptierenden Vorgänger zur aktuellen Menge von akzeptierenden Markierungen $\mathcal{Y}$ berechnet. Dies wird solange wiederholt, bis sich die Markierungsmenge $\mathcal{Y}$ nicht mehr verändert, also keine akzeptierende Markierung mehr entfernt wird. Die äußere Fixpunktiteration erzeugt solange eine absteigende Inklusionskette von Markierungsmengen, bis ein Fixpunkt erreicht wird. Bezeichne $\mathcal{Y}_i$ den i-ten Iterationsschritt der äußeren Fixpunktiteration, dann gilt für ein $k \in \mathbb{N}$:

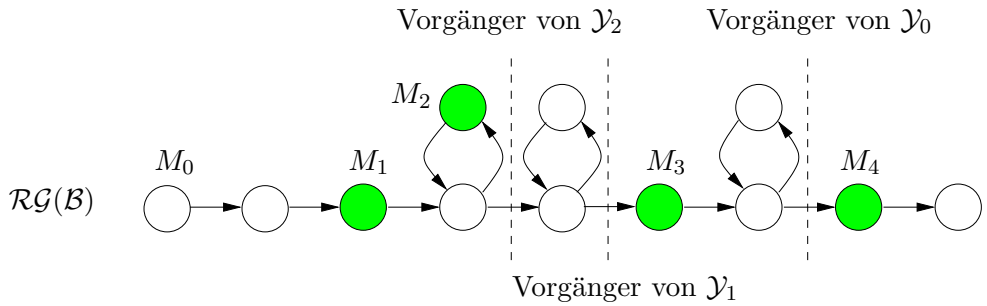
$$\mathcal{F} = \mathcal{Y}_0 \supset \mathcal{Y}_1 \supset \cdots \supset \mathcal{Y}_k = \mathcal{Y}_{k+1}.$$

Algorithmus 17 (BwdCycleCheck)

```
1  BwdCycleCheck([[S, T, F, M_0], S_F]) ≡
2    F := ⋁_{s_i ∈ S_F} s_i;
3
4    func BwdFixP(M) ≡
5      Y' := M;
6      repeat
7        Y := Y';
8        Z' := ∅;
9        repeat
10         Z := Z';
11         Z' := prev(Z ∪ Y)
12       until Z = Z';
13       Y' := Z' ∩ F
14     until Y = Y';
15     return (Z, Y)
16   end BwdFixP;
17
18 begin
19   (Z, Y) := BwdFixP(F);
20   if (Y ≠ ∅) ∧ (M_0 ∈ Z) then
21     exit("accepting cycle found")
22   else
23     exit("no accepting cycle found")
24   fi
25 end.
```

Ist die Markierung nach dem Erreichen des äußeren Fixpunktes nicht leer, so existiert mindestens eine akzeptierende Markierung $M \in \mathcal{Y}$, die sich selber erreichen kann. Ist der Initialzustand M_0 in der Vorgängermenge $\mathcal{Z}$ von $\mathcal{Y}$ enthalten, so ist diese Markierung M auch von M_0 aus erreichbar. Es existiert also ein akzeptierender Systemablauf unseres Produktbüchinetzes.

Beispiel 13 (BwdCycleCheck($\mathcal{B}$))



Der Startwert von $\mathcal{Y}$ ist $\mathcal{Y}_0 = \{M_1, M_2, M_3, M_4\}$. Nach der ersten inneren Fixpunktiteration erhalten wir $\mathcal{Y}_1 = \{M_1, M_2, M_3\}$. Nach der zweiten inneren Fixpunktiteration

ist $\mathcal{Y}_2 = \{M_1, M_2\}$. Bei der dritten inneren Fixpunktiteration tritt keine Veränderung mehr auf, so daß wir einen Fixpunkt der äußeren Fixpunktiteration erreicht haben.

Die in diesem Algorithmus verwendete Rückwärtsiteration ist, wie die Arbeiten [TBK95, INH96, HKSV97] belegen, oft sehr zeitaufwendig. Die Bestimmung aller möglichen Vorgängermarkierungen einer Zustandsmenge erzeugt auch nicht erreichbare Markierungen des Büchinetzes. Das führt oft zu einer großen Anzahl von Iterationsschritten und einer größeren BDD-Darstellung der Markierungsmengen.
Es gibt daher einige Ansätze, diesen Effekt zu verhindern oder abzuschwächen. Ein von K. Lautenbach und H. Ridder in [LR95] verwendetes Verfahren approximiert die tatsächliche Zustandsmenge durch eine im voraus durchgeführte Berechnung von Strukturinvarianten des Petrinetzes, wie z. B. die schon bekannten Stelleninvarianten. Selbst die Berechnung der Menge der erreichbaren Zustände, als einen zusätzlichen Schritt vor der Rückwärtsiteration, ist oft schneller als das direkte Verfahren [Rid97].
Die beste Möglichkeit mit dem Problem fertig zu werden, ist allerdings die Verwendung eines äquivalenten, auf Vorwärtsiteration basierenden Verfahrens. Der Algorithmus 18 (FwdCycleCheck) ist eine Implementierung eines solchen Verfahrens.

Algorithmus 18 (FwdCycleCheck)

```
1  FwdCycleCheck([[S,T,F,M_0],S_F]) ≡
2    F := ⋁_{s_i∈S_F} s_i;
3
4    func FwdFixP(M) ≡
5      Y' := M;
6      repeat
7        Y := Y';
8        Z' := ∅;
9        repeat
10         Z := Z';
11         Z' := next(Z ∪ Y)
12       until Z = Z';
13       Y' := Z' ∩ F
14     until Y = Y';
15     return Y
16   end FwdFixP;
17
18 begin
19   Y := FwdFixP({M_0});
20   if Y ≠ ∅ then
21     exit("accepting cycle found")
22   else
23     exit("no accepting cycle found")
24   fi
25 end.
```

Dieser Algorithmus besteht ebenfalls aus zwei ineinander geschachtelten Fixpunktiterationen. Er unterscheidet sich gegenüber dem Algorithmus 17 (BwdCycleCheck) dadurch, daß er in umgekehrter Richtung arbeitet, also diejenigen akzeptierenden Markierungen aus $\mathcal{Y}$ entfernt, die keinen Vorgänger mehr in $\mathcal{Y}$ besitzen. Dabei betrachtet er nur die vom Büchinetz erreichbaren Zustände.
Es war lange ein offenes Problem, ob LTL-Modelchecking in einem Fixpunktkalkül, welches nur Vorwärtsiteration zuläßt, ausdrückbar ist. Dies wurde erst 1998 in der Arbeit von T. A. Henzinger, O. Kupferman und S. Qadeer [HKQ98] positiv beantwortet. Der Algorithmus 18 (FwdCycleCheck) wurde allerdings schon 1997 auf einem Workshop [Spr97a] vorgestellt. Der Zusammenhang mit dem LTL-Modelchecking wurde damals aber nicht erkannt.
Die beiden vorgestellten Algorithmen ergeben zusammen mit der Produktbüchinetzkonstruktion ein vollständiges Modelcheckingverfahren für LTL-Spezifikationen. Die Entscheidung über die Erfüllbarkeit einer LTL-Spezifikation ist also erst am Ende der Algorithmen, also wenn keine akzeptierenden Markierungen mehr entfernt werden, möglich. Die Algorithmen betrachten also immer alle auf einem Kreis liegenden akzeptierenden Zustände des Büchinetzes, obwohl uns die Existenz eines solchen Zustandes eigentlich genügen würde. Sie eignen sich daher nicht für ein on-the-fly Verifikationsverfahren. Weiterhin ist der zeitliche Aufwand der Algorithmen quadratisch bzgl. der Größe des Erreichbarkeitgraphen unseres Produktbüchinetzes, wodurch sie nicht die gleiche zeitliche Aufwandsklasse der Verfahren aus dem Abschnitt 5.4 erreichen.

Satz 26 (Komplexität der Algorithmen)
Die Überprüfung, ob ein akzeptierender Lauf im Erreichbarkeitsgraphen eines Büchinetzes $\mathcal{B} = [[S, T, F, M_0], S_F]$ existiert, ist

1. *mit dem Algorithmus FwdCycleCheck im schlimmsten Fall mit einem zeitlichen Aufwand von $\mathcal{O}(|\mathcal{R}_{\mathcal{B}}(M_0)|^2)$ und*

2. *mit dem Algorithmus BwdCycleCheck mit einem zeitlichen Aufwand von $\mathcal{O}((2^{|S|})^2)$ entscheidbar.*

Beweis: Betrachten wir den Algorithmus FwdCycleCheck. Die innere Fixpunktiteration benötigt maximal $|\mathcal{R}_{\mathcal{B}}(M_0)|$ *next*-Operationen zur Berechnung aller Nachfolger einer gegebenen Zustandsmenge. Die äußere Fixpunktiteration verringert die gegebenen akzeptierenden Zustände bei jeder Iteration um mindestens einen Zustand, so daß sie ebenfalls maximal $|\mathcal{R}_{\mathcal{B}}(M_0)|$ *next*-Operationen benötigt. Dies führt zu einem zeitlichen Aufwand von $\mathcal{O}(|\mathcal{R}_{\mathcal{B}}(M_0)|^2)$.
Als nächstes zeigen wir, daß die Aufwandsklasse auch tatsächlich erreicht wird. O. B. d. A. sei die Anzahl der erreichbaren Zustände von $\mathcal{B}$ gerade. Betrachten wir für den Algorithmus FwdCycleCheck einen Erreichbarkeitsgraphen, dessen erreichbare Zustände zur Hälfte aus akzeptierenden Zuständen bestehe. Diese seien so angeordnet, daß die

akzeptierenden Zustände eine Kette bilden und die nicht akzeptierenden Zustände mit dem letzten akzeptierenden Zustand der Kette einen Kreis ergeben. Für einen solchen Erreichbarkeitsgraphen benötigt die innere Fixpunktiteration zur Erzeugung der nicht akzeptierenden Zustände immer $\frac{|\mathcal{R}_{\mathcal{B}}(M_0)|}{2} + 1$ *next*-Operationen. In der äußeren Fixpunktiteration wird bei jedem Durchlauf immer nur ein akzeptierender Zustand entfernt. Daher wird sie $\frac{|\mathcal{R}_{\mathcal{B}}(M_0)|}{2}$-mal durchlaufen. Man benötigt also insgesamt $\frac{|\mathcal{R}_{\mathcal{B}}(M_0)|}{2} \cdot (\frac{|\mathcal{R}_{\mathcal{B}}(M_0)|}{2} + 1)$ *next*-Operationen, was zu einem zeitlichen Aufwand von $\mathcal{O}(|\mathcal{R}_{\mathcal{B}}(M_0)|^2)$ führt.
Betrachten wir nun den Algorithmus BwdCycleCheck. Durch die verwendete Vorgängerberechnung besteht die Möglichkeit, daß durch die innere Fixpunktiteration immer alle möglichen $2^{|S|}$ Markierungen des Büchinetzes erzeugt werden. Da die äußere Fixpunktiteration die gegebenen akzeptierenden Zustände bei jeder Iteration um mindestens einen Zustand verringert, benötigt sie maximal $2^{|S|}$ Iteration. Daraus ergibt sich dann der zeitliche Aufwand von $\mathcal{O}((2^{|S|})^2)$.
Berechnet man im voraus alle erreichbaren Zustände und schränkt die Vorgängerfunktion darauf ein, erreicht man mit dem Algorithmus BwdCycleCheck dieselbe Aufwandsklasse wie der FwdCycleCheck Algorithmus. □

6.3 Symbolische Tiefensuche

Unser Modelcheckingproblem haben wir auf das Auffinden von akzeptierenden Kreisen im Erreichbarkeitsgraphen eines entsprechenden Produktbüchinetzes reduziert.
Die klassischen Verifikationsverfahren betrachten jeden Zustand und jede mögliche Zustandsabfolge des Erreichbarkeitsgraphen separat. Die Zustandsabfolgen werden darauf untersucht, ob sich auf einem durch einen wiederholenden Zustand beschriebenen Kreis akzeptierende Zustände befinden.
Überträgt man diesen Ansatz auf Zustandsmengen und die Nachfolgerfunktion *next*, konstruiert also eine bei $\{M_0\}$ beginnende Folge von Zustandsmengen, so läßt sich nach dem wiederholten Auftreten einer Zustandsmenge $\mathcal{M}$ nur dann auf einen akzeptierenden Kreis im Erreichbarkeitsgraphen schließen, wenn eine Zustandsmenge auf dem beschriebenen Kreis nur aus akzeptierenden Zuständen besteht.

Satz 27
Sei gegeben eine Folge $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k$, $k \in \mathbb{N}$ von Zustandsmengen eines Büchinetzes $\mathcal{B}$ mit $\mathcal{A}_{i+1} = next(\mathcal{A}_i)$, $0 \leq i \leq k-1$ und $\mathcal{A}_0 = \mathcal{A}_k$, dann gilt: $\exists\, M \in \mathcal{A}_0 \colon M \xrightarrow{*} M$.

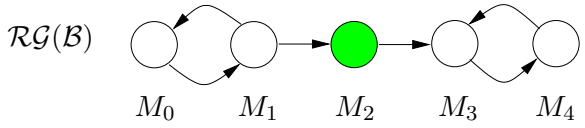
Beweis: Sei n die Anzahl der Zustände von $\mathcal{A}_0$. Jeder Zustand $M_1 \in \mathcal{A}_k$ hat per Konstruktion einen Vorgänger M_2 in der Zustandsmenge $\mathcal{A}_0$, also gilt: $M_2 \xrightarrow{*} M_1$. Ebenso hat M_2 wiederum einen Vorgänger M_3 in $\mathcal{A}_0$, woraus $M_3 \xrightarrow{*} M_2 \xrightarrow{*} M_1$ folgt. Dies können wir n-mal wiederholen und erhalten damit eine Zustandsfolge

$$M_{n+1} \xrightarrow{*} M_n \xrightarrow{*} \cdots \xrightarrow{*} M_1$$

bestehend aus $n+1$ Zuständen. Da $\mathcal{A}_0$ nur n Zustände enthält, müssen mindestens zwei dieser $n+1$ Zustände identisch sein. Sie bilden also einen Kreis. □

Der Satz 27 besagt nicht, daß alle Zustände aus einer sich wiederholenden Zustandsmenge $\mathcal{A}$ auf einem Kreis im Erreichbarkeitsgraphen liegen. Dies verdeutlicht das folgende Beispiel.

Beispiel 14



Für den gegebenen Erreichbarkeitsgraphen $\mathcal{RG}(\mathcal{B})$ ergibt sich die Zustandsmengenfolge:

- $\mathcal{A}_0 := \{M_0\}$
- $\mathcal{A}_1 := \{M_1\} = next(\mathcal{A}_0)$
- $\mathcal{A}_2 := \{M_0, M_2\} = next(\mathcal{A}_1)$
- $\mathcal{A}_3 := \{M_1, M_3\} = next(\mathcal{A}_2)$
- $\mathcal{A}_4 := \{M_0, M_2, M_4\} = next(\mathcal{A}_3)$
- $\mathcal{A}_5 := \{M_1, M_3\} = next(\mathcal{A}_4) = \mathcal{A}_3$

Der akzeptierende Zustand M_2 ist zwar in einer sich wiederholenden Zustandsmenge enthalten, liegt aber auf keinem Kreis im Erreichbarkeitsgraphen.

Wollen wir also mit Zustandsmengen arbeiten, müssen wir auf eine Trennung zwischen akzeptierenden und nicht akzeptierenden Zuständen achten. Dies kann man dadurch erreichen, daß man die durch eine *next*-Operation erzeugte Nachfolgerzustandsmenge in eine Menge mit akzeptierenden Zuständen und in eine Menge mit nicht akzeptierenden Zustände unterteilt. Danach betrachtet man die beiden Nachfolgermengen separat. Man erhält somit einen Graphen, dessen Knoten aus Zustandsmengen bestehen (s. Abbildung 6.2). Als Invariante für jeden Knoten K des Graphen gilt:

1. Die Zustandsmengen aller direkten Nachfolgeknoten sind disjunkt.
2. Eine Zustandsmenge eines Knotens besteht entweder nur aus akzeptierenden oder nur aus nicht akzeptierenden Zuständen.

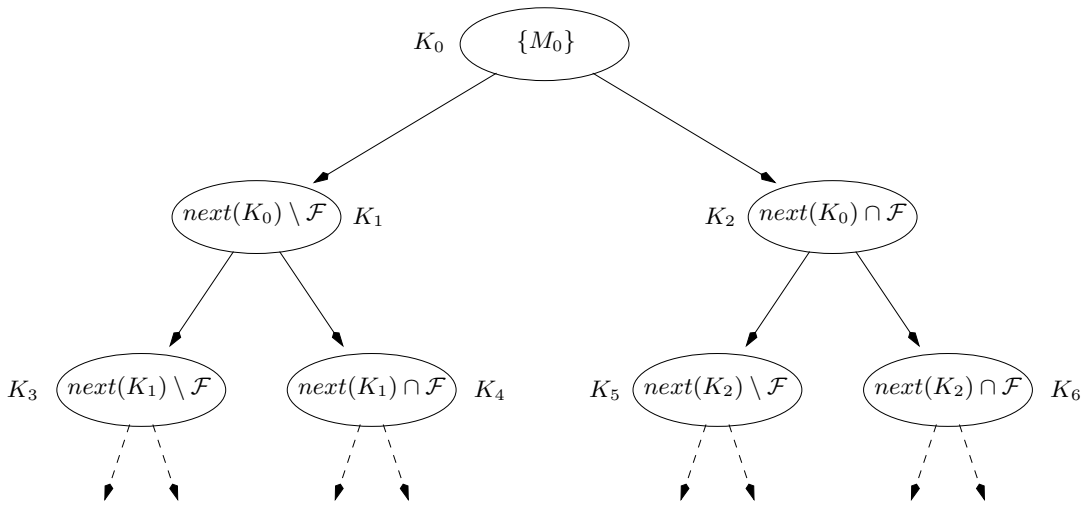


Abbildung 6.2: Beispielgraph zur Darstellung der Zerlegung der Nachfolgemarkierungen in Mengen akzeptierender und nicht akzeptierender Zustände.

3. Die Vereinigung aller direkten Nachfolgeknoten von K ist gleich der Zustandsmenge $next(K)$.

Zur Überprüfung eines solchen Graphen auf Kreise, die einen Knoten, bestehend aus akzeptierenden Zuständen enthalten, lassen sich die im Kapitel 5 vorgestellten Verfahren verwenden.
Diese besitzen, wie wir wissen, einen linearen zeitlichen Aufwand bezüglich der Graphengröße.
Die maximale Anzahl der möglichen Knoten des Graphen entspricht der Anzahl der unterschiedlichen Teilmengen von $\mathcal{R}_{\mathcal{B}}(M_0)$. Sie ist daher im schlimmsten Fall exponentiell bzgl. der Anzahl der Zustände von $\mathcal{R}_{\mathcal{B}}(M_0)$.
Der Grund für die hohe Anzahl von möglichen Knoten liegt darin, daß Zustände mehrmals repräsentiert sein können, sich also Zustandsmengen überlappen.
Würde keine Überlappung der Zustandsmengen vorkommen, so würden die Zustandsmengen eine Partitionierung von $\mathcal{R}_{\mathcal{B}}(M_0)$ ergeben und der Graph somit aus maximal $|\mathcal{R}_{\mathcal{B}}(M_0)|$ Knoten bestehen.
Wir erreichen eine solche Partitionierung, indem wir beim Hinzufügen eines neuen Knotens mit der Zustandsmenge $\mathcal{M}$ folgendermaßen verfahren. Sei $\mathcal{U}$ die Menge der im Graphen schon vorhandenen Zustandsmengen. Existiert eine Zustandsmenge $\mathcal{M}' \in \mathcal{U}$ mit $\mathcal{M} \cap \mathcal{M}' \neq \emptyset$, so entfernen wir $\mathcal{M}'$ aus $\mathcal{U}$ und versuchen anschließend die Zustandsmengen $\mathcal{M} \setminus \mathcal{M}'$, $\mathcal{M} \cap \mathcal{M}'$ und $\mathcal{M}' \setminus \mathcal{M}$ der Menge $\mathcal{U}$ hinzuzufügen. Existiert keine Überlappung von $\mathcal{M}$ mit einer Menge aus $\mathcal{U}$, so setzen wir $\mathcal{U} := \mathcal{U} \cup \{\mathcal{M}\}$.
Durch dieses Vorgehen erhält man zwar eine Beschränkung der Knotenzahl des Gra-

phen auf $|\mathcal{R}_{\mathcal{B}}(M_0)|$, doch leider ist der Ansatz so nicht mit unseren Tiefensucheverfahren verträglich. Beim Hinzufügen einer neuen Zustandsmenge können schon früher erzeugte Zustandsmengen aufgeteilt werden, was eine ungewollte Neuberechnung für die jeweiligen Teilmengen nach sich ziehen würde.
Das Verändern schon erzeugter Zustandsmengen kann man dadurch verhindern, daß man auch das Hinzufügen von echten Teilmengen schon vorhandener Zustandsmengen erlaubt.
Dies verdoppelt maximal die Knotenanzahl, was der folgende Satz belegt.

Satz 28
Sei eine Menge $\mathcal{V}$ mit $|\mathcal{V}| \geq 1$ gegeben. Für jede Menge $\mathcal{U} \subseteq 2^{\mathcal{V}}$ mit $\forall U_i, U_j \in \mathcal{U}$: $(U_i \cap U_j = \emptyset \vee U_i \subset U_j \vee U_j \subset U_i)$ gilt: $|\mathcal{U}| \leq 2 \cdot |\mathcal{V}|$.

Beweis: Es genügt, wenn wir die Gültigkeit der Ungleichung für eine beliebige, aber feste, die Einschränkungen erfüllende Menge $\mathcal{U}$ mit maximal vielen Elementen zeigen.
Die Menge $\mathcal{U}$ enthält wegen der Maximalität ihrer Elementanzahl die Elemente $\emptyset$ und $\mathcal{V}$. Die Elemente von $\mathcal{U}$ bilden bzgl. der Mengeninklusion „$\subset$“ eine partielle Ordnung, wobei $\emptyset$ das kleinste Element und $\mathcal{V}$ das größte Element bilden. Für alle Elemente $U \in \mathcal{U}$ mit $|U| > 1$, bilden die direkten Vorgänger $U_1, \ldots, U_k$ bzgl. der partiellen Ordnung eine vollständige Partitionierung von U. Wegen der Vollständigkeit der Partitionierung und der Inklusionsbeziehung gilt für die Anzahl der Vorgänger einer solchen Menge $k \geq 2$.
Wir bezeichnen im folgenden mit $A(\mathcal{V})$ die maximale Anzahl von Elementen einer Menge $\mathcal{U} \subseteq 2^{\mathcal{V}}$ mit den geforderten Eigenschaften.

1. Für $|\mathcal{V}| = 1$ ergibt sich: $A(\mathcal{V}) = |\{\mathcal{V}\}| + |\{\emptyset\}| = 2$.

2. Für $|\mathcal{V}| \geq 2$ gilt:

   $$A(\mathcal{V}) = |\{\mathcal{V}\}| + |\{\emptyset\}| + A(\mathcal{V}_1) + \cdots + A(\mathcal{V}_k) - k \cdot |\{\emptyset\}| = 2 + A(\mathcal{V}_1) + \cdots + A(\mathcal{V}_k) - k$$

   mit $\mathcal{V}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{V}_k = \mathcal{V}$ und $k \geq 2$. Da die leere Menge in allen k Partitionen enthalten ist, müssen wir k abziehen.

Wir zeigen nun durch Induktion über der Anzahl der Elemente von $\mathcal{V}$, daß $A(\mathcal{V}) \leq 2 \cdot |\mathcal{V}|$ gilt.

IA $(|\mathcal{V}| = 1)$: $A(\mathcal{V}) = 2 \leq 2 \cdot |\mathcal{V}|$.

IS ($|\mathcal{V}| = n+1$): Es sei die Ungleichung für $|\mathcal{V}| \leq n$ schon gezeigt (IH). Dann gilt:

$$\begin{array}{rcl} A(\mathcal{V}) & = & 2 + A(\mathcal{V}_1) + \cdots + A(\mathcal{V}_k) - k \\ & \underset{(k\geq 2)}{\leq} & A(\mathcal{V}_1) + \cdots + A(\mathcal{V}_k) \\ & \underset{(\mathsf{IH})}{\leq} & 2 \cdot |\mathcal{V}_1| + \cdots + 2 \cdot |\mathcal{V}_k| \\ & = & 2 \cdot (|\mathcal{V}_1| + \cdots + |\mathcal{V}_k|) \\ & = & 2 \cdot |\mathcal{V}| \end{array}$$

□

Basierend auf diesem Ergebnis verändern wir die Graphkonstruktion, indem wir nach dem Erzeugen eines Knotens diesen mit der Funktion

$$\begin{array}{l} \mathit{divide}\colon 2^{(2^{\mathcal{R}_{\mathcal{B}}(M_0)})} \times 2^{\mathcal{R}_{\mathcal{B}}(M_0)} \longrightarrow 2^{(2^{\mathcal{R}_{\mathcal{B}}(M_0)})}, \\ \mathit{divide}(\mathcal{U}, \mathcal{M}) := \{\, \mathcal{M}_1, \ldots, \mathcal{M}_k \mid \mathcal{M}_1 \,\dot\cup \cdots \dot\cup\, \mathcal{M}_k = \mathcal{M} \wedge \\ \qquad (\forall\, \mathcal{M}' \in \mathcal{U}\colon \mathcal{M}' \cap \mathcal{M}_i = \emptyset \vee \mathcal{M}' \subseteq \mathcal{M}_i \vee \mathcal{M}_i \subseteq \mathcal{M}', 1 \leq i \leq k) \,\} \end{array}$$

bezüglich den schon existierenden Knoten $\mathcal{U}$ unterteilen.

Beispiel 15 (*divide*)
Sei die Zustandsmenge $\mathcal{U} = \{\, \{M_0\}, \{M_1, M_2\}, \{M_4, M_5\} \,\}$ und die Menge der Nachfolgemarkierungen von $\{M_1, M_2\}$ als $\mathit{next}(\{M_1, M_2\}) = \{M_0, M_4, M_6, M_7\}$ gegeben. Die Funktion *divide* unterteilt $\{M_0, M_4, M_6, M_7\}$ bzgl. $\mathcal{U}$ unter Beachtung von Satz 28 in die Zustandsmenge $\{\, \{M_0\}, \{M_4\}, \{M_6, M_7\} \,\}$.

Die durch unser abgewandeltes Verfahren konstruierbaren Graphen bezeichnen wir im folgenden als *Mengengraphen* $\mathcal{MG}(\mathcal{B})$ eines zugrundeliegenden Büchinetzes $\mathcal{B}$. Die Konstruktion eines Mengengraphen ist nicht eindeutig, da die Knoten und damit die Zustandsmengen von der Reihenfolge des Hinzufügens der Zustandsmengen abhängig sind.
Nichtdestotrotz besitzen alle Mengengraphen die Eigenschaften:

1. Alle Nachfolgeknoten eines Knoten K des Graphen sind entweder disjunkt zu K oder eine Teilmenge von K oder aber K ist eine Teilmenge eines solchen Knotens.

2. Jeder Knoten besteht entweder nur aus akzeptierenden oder nur aus nicht akzeptierenden Zuständen.

3. Die Vereinigung aller direkten Nachfolgeknoten eines Knoten K im Graphen ist gleich der Zustandsmenge $\mathit{next}(K)$.

Für alle zu einem Büchinetz konstruierbaren Mengengraphen gilt:

Satz 29
Es sei $\mathcal{MG}(\mathcal{B})$ ein Mengengraph des Büchinetzes $\mathcal{B} = [[S, T, F, M_0], S_f]$. Ein akzeptierender Lauf σ im Erreichbarkeitsgraphen von $\mathcal{B}$ existiert genau dann, wenn eine Knotenfolge $K_0, \ldots, K_i, \ldots, K_j$, $i \neq j$ in $\mathcal{MG}(\mathcal{B})$ existiert mit $K_0 = \{M_0\}$, K_i besteht aus akzeptierenden Zuständen von $\mathcal{R}_B(M_0)$ und $K_i = K_j$.

Beweis: „⇒“: Sei o. B. d. A. der Zustand M ein akzeptierender Zustand, der sich im Lauf σ unendlich oft wiederholt. Entrollt man den Mengengraphen $\mathcal{MG}(\mathcal{B})$, beginnend beim Knoten $K_0 = \{M_0\}$, so erhält man einen Baum. Die Vereinigung der Knoten einer Ebene i entspricht dabei dem Ergebnis der i-fachen Anwendung der *next*-Operation auf $\{M_0\}$. Daher erreicht man nach endlich vielen Schritten einen Knoten K_1 aus $\mathcal{MG}(\mathcal{B})$ mit $M \in K_1$. Betrachtet man den Teilbaum mit Wurzelknoten K_1, so gilt mit der gleichen Argumentation, daß ein Knoten K_2 mit $M \in K_2$ von K_1 aus erreichbar ist. Für die beiden Knoten K_1 und K_2 gilt aufgrund des Konstruktionsverfahrens des Mengengraphen $K_1 \supseteq K_2$. Diese Argumentation läßt sich beliebig oft wiederholen, so daß wir eine unendliche Knotenfolge mit $K_1 \supseteq K_2 \supseteq \cdots \supseteq K_i \supseteq \cdots$ konstruieren können. Da es nur endlich viele echte Teilmengen der Zustandsmenge des Knoten K_1 gibt, müssen sich Knoten der Folge wiederholen.
„⇐“: Aufgrund von Satz 27 existiert für eine Knotenfolge $K_0, \ldots, K_i, \ldots, K_j$, mit den geforderten Eigenschaften, ein Zustand $M \in K_i$, der auf einem Kreis liegt. Da die Zustände der Knoten eines Mengengraphen alles Nachfolger der Zustände des Wurzelknotens K_0 sind, ist M von M_0 aus erreichbar. □

Die Suche nach akzeptierenden Kreisen in einem Mengengraphen kann mit einem der bekannten Tiefensucheverfahren erfolgen. Wir wollen daher im weiteren ein solches Modelcheckingverfahren als *symbolische Tiefensuche* bezeichnen.
Eine Implementierung der symbolischen Tiefensuche ist im Algorithmus 19 (Symbolic-DFS) dargestellt. Es handelt sich dabei um eine Modifikation des im Kapitel 5 betrachteten Tarjan-Algorithmus zur Berechnung stark-zusammenhängender Komponenten (vgl. Algorithmus 14 (SZK)).
Die im Algorithmus verwendete Funktion divide ist eine Implementierung der oben definierten Funktion *divide*.
Zur Speicherung der in den Knoten vorkommenden Zustandsmengen wird eine gemeinsame Datenstruktur, die sogenannten Shared BDDs verwendet (vgl. Abschnitt 3.2.5). Dies ermöglicht die gemeinsame Nutzung gleicher Untergraphen der den einzelnen Zustandsmengen zugeordneten BDDs.
Die bei der Tiefensuche erfolgende Auswahl des als nächsten zu betrachtenden Nachfolgerknotens kann durch eine einfache Heuristik ergänzt werden. Da wir uns für das schnelle Auffinden von akzeptierenden Kreisen im Graphen interessieren, betrachten wir zuerst die Nachfolgermengen, die aus akzeptierenden Zuständen bestehen.

Algorithmus 19 (SymbolicDFS)

```
1  SymbolicDFS([[S, T, F, M_0], S_F]) ≡
2    U := ∅;
3    Stack := ∅;
4    count := 0;
5    F := ⋁_{s_i ∈ S_F} s_i;
6
7    func divide(U, M) ≡
8      tmp := ∅;
9      forall M' ∈ U do
10       if M ⊆ M' then tmp := tmp ∪ {M}
11       elsif M ⊃ M' then
12         tmp := tmp ∪ {M'};
13         M := M \ M'
14       elsif M ∩ M' ≠ ∅ then
15         tmp := tmp ∪ {M ∩ M'};
16         M := M \ M'
17       fi
18     od;
19     if M ≠ ∅ then return tmp ∪ M
20     else return tmp
21     fi
22   end divide;
23
24   proc visit(M) ≡
25     U := U ∪ {M};
26     count := count + 1;
27     M.num := count;
28     M.uplink := count;
29     Stack := Stack ∪ {M};
30     tmp := next(M);
31     forall M' ∈ divide(U, tmp ∩ F) ∪ divide(U, tmp \ F) do
32       if M' ∉ U then
33         visit(M');
34         M.uplink := min(M.uplink, M'.uplink)
35       elsif M' ∈ Stack then
36         M.uplink := min(M.uplink, M'.num)
37       fi
38     od;
39     if M.num = M.uplink then
40       szk := { M' | M' ∈ Stack ∧ M'.num ≥ M.num };
41       if ∃ M' ∈ szk: M' ∩ F ≠ ∅ then
42         exit("accepting SZK found")
43       fi;
44       Stack := Stack \ szk
45     fi
46   end visit;
47
48 begin
49   visit({M_0});
50   exit("no accepting SZK found")
51 end.
```

Satz 30 (Komplexität der symbolischen Tiefensuche)
Die Überprüfung, ob ein akzeptierender Lauf im Erreichbarkeitsgraphen eines Büchinetzes $\mathcal{B}$ existiert, ist durch die symbolische Tiefensuche mit einem zeitlichen Aufwand von $\mathcal{O}(|\mathcal{RG}(\mathcal{B})|)$ entscheidbar.

Beweis: Nach Satz 29 ist die Suche nach einem akzeptierenden Lauf auf ein Erreichbarkeitsproblem in einem Mengengraphen von $\mathcal{B}$ zurückführbar. Einen Kreis im Mengengraphen zu finden, der einen Knoten mit akzeptierenden Zuständen enthält, ist mit dem Algorithmus 19 (SymbolicDFS) mit linearem Aufwand bzgl. der Mengengraphengröße möglich [Tar72]. Nach Satz 28 ist die Knotenzahl eines Mengengraphen durch das Doppelte der Knotenzahl der erreichbaren Markierungen von $\mathcal{B}$ beschränkt. □

Mit der symbolischen Tiefensuche haben wir also ein Verfahren mit derselben zeitlichen Aufwandsklasse wie die im Kapitel 5 vorgestellten Verfahren. Des weiteren erlaubt das Verfahren eine on-the-fly Verifikation, da nach dem Auftreten eines akzeptierenden Kreises im Mengengraphen die Verifikation, und damit der weitere Aufbau des Mengengraphen, abgebrochen werden kann.

6.4 Zusammenfassende Bemerkungen

Der Büchinetzformalismus wurde von J. Esparza und S. Melzer [EM97] eingeführt, um basierend auf strukturellen Eigenschaften des Produktbüchinetzes, also durch Umgehung der Zustandsraumexplosion, Aussagen über die Erfüllbarkeit von LTL-Spezifikationen zu machen. Sie erhielten dadurch ein schnelles Halbentscheidungsverfahren für spezielle LTL-Spezifikationen.
Möchte man die volle LTL abdecken, kommt man an einer Generierung des Erreichbarkeitsgraphen des Produktbüchinetzes nicht vorbei. Um dennoch der großen Anzahl von Zuständen Herr zu werden, wurden hier binäre Entscheidungsgraphen verwendet. Als vorteilhaft erweist sich dabei die Möglichkeit, mit Hilfe der Büchinetze sowohl das Systempetrinetz als auch den Formelbüchiautomaten mit dem gleichen Formalismus behandeln zu können. Die Betrachtung des Gesamtsystems als eine spezielle Art von Petrinetzen ermöglicht es uns, vom Modelcheckingproblem auf ein Erreichbarkeitsproblem für Petrinetze zu abstrahieren. Damit ist es auf einfache Weise möglich, die im Kapitel 4 auf der Berechnung von 1-S-Invarianten eingeführten Verfahren zu einer effizienten Behandlung von Petrinetzen mit BDDs auf das gesamte Produktbüchinetz anzuwenden.
Ein symbolisches Verfahren zum LTL-Modelchecking wurde auch schon in der Arbeit von E. M. Clarke, O. Grumberg und K. Hamaguchi [CGH94] beschrieben. Sie führten darin das LTL-Modelcheckingproblem auf eine Fragestellung in einer anderen temporalen Logic, der *Computational Tree Logic* (kurz CTL) mit Fairness-Bedingungen,

zurück. Man landet mit diesem Ansatz im Endeffekt bei einer Version des bekannten Emerson-Lei-Verfahrens. Die Probleme des Emerson-Lei-Verfahrens beruhen auf der Erzeugung nicht erreichbarer Markierungen. Diese können zu einer hohen Anzahl von Berechnungsschritten und einem hohen Speicherplatzbedarf führen. Das in diesem Kapitel eingeführte Vorwärtsiterationsverfahren arbeitet nur auf den erreichbaren Zuständen des Produktbüchinetzes und umgeht damit die Problematik des Emerson-Lei-Verfahrens. Des weiteren gibt es eine Antwort auf die Fragestellung, ob das LTL-Modelcheckingproblem in einem nur Vorwärtsiteration zulassenden Fixpunktkalkül ausdrückbar ist (vgl. [HKQ98]). Beide Verfahren arbeiten bestenfalls mit einem quadratischen Aufwand bezüglich der Anzahl von erreichbaren Zuständen des Petrinetzes und eignen sich auch nicht für ein anzustrebendes on-the-fly Verifikationsverfahren.
Aus diesem Grund wurde die symbolische Tiefensuche entwickelt. Sie stellt eine Mischung aus den klassischen Tiefensucheverfahren und der symbolischen Behandlung von Zustandsräumen dar. Es kombiniert die Vorteile der beiden Ansätze und bietet ein on-the-fly Verifikationsverfahren mit linearem zeitlichem Aufwand und zugleich eine symbolische Darstellung der Zustandsmengen. Der sich daraus ergebende Anwendungsbereich ist daher auch zwischen den ursprünglichen Methoden anzusiedeln, also in der Designphase eines Systems, wo eine häufige Überprüfung der Eigenschaften vorgenommen wird und man mit Fehlern zu rechnen hat, dies aber mit den klassischen Verfahren wegen der Zustandsraumgröße nur schwer handhabbar ist.
Wichtig im Fehlerfall ist die Erzeugung eines Gegenbeispiels. Dies läßt sich mit der symbolischen Tiefensuche einfach bewerkstelligen, indem man ein allgemeines symbolisches Verfahren zur Bestimmung von Gegenbeispielen (vgl. z. B. [KPR98]) auf die im Stapel abgelegten Zustandsmengen beschränkt, was die Erzeugung eines Gegenbeispiels beschleunigt.

7 Abschlußbemerkungen

7.1 Zusammenfassung

Das angestrebte Ziel der vorliegenden Arbeit war es, die drei Bereiche: Petrinetze als Modellierungssprache, linear-time temporale Logik zur Spezifikation von Systemeigenschaften und binäre Entscheidungsgraphen zur Repräsentation großer Zustandsmengen zu einem effizienten Modelcheckingverfahren zu vereinen.
Dabei wurde darauf geachtet, die Vorteile der einzelnen Bereiche im entstehenden Verfahren beizubehalten und die sich zwischen den Bereichen ergebenden Synergien zu nutzen.
Wir haben uns in dieser Arbeit auf sichere Stellen/Transitions-Netze beschränkt, da sie einfach handhabbar sind und, wie viele Analysewerkzeuge zeigen, eine praktikable Modellierungssprache für eine Vielzahl von Systemen darstellen. Basierend auf der Struktur eines Petrinetzes lassen sich Aussagen über die mögliche Verteilung von Marken auf den Stellen für alle erreichbaren Zustände des Petrinetzes treffen. Die eingeführten 1-Stellen-Invarianten repräsentieren Stellenmengen, bei denen in jedem erreichbaren Zustand des Petrinetzes genau eine Stelle markiert ist. Diese strukturellen Informationen wurden anschließend zur Unterstützung einer kompakten Darstellung und effizienten Handhabung von Zuständen benutzt.
Zur Handhabung von großen Zustandsmengen wurden binäre Entscheidungsgraphen verwendet. Die Zustandsdarstellung und die Petrinetzschaltsemantik wurden dazu auf ROBDDs und ROBDD-Operationen übertragen.
Die Größe der ROBDD-Darstellung der Zustände ist dabei für die Effizienz des Ansatzes maßgeblich. Sie hängt neben der nicht beeinflußbaren Komplexität des vorgegebenen Systems von der zur Codierung der Zustände benötigten Variablenanzahl und der Variablenreihenfolge ab. Daher wurden zwei Ansätze entwickelt, die sich diesem Problem widmen. Sie basieren beide auf den 1-S-Invarianten des zugrundeliegenden Petrinetzes. Das erste Verfahren nutzt die 1-S-Invarianten zur Erzeugung einer günstigen Variablenordnung. Dazu werden die Variablen zu Blöcken geordnet, die zu kleinen lokalen ROBDD-Darstellungen führen, was zu einer Reduktion der Gesamtgröße der Darstellung führt. Das zweite Verfahren nutzt die Eigenschaft, daß zwei Stellen einer 1-S-Invariante nicht gleichzeitig markiert sein können. Dadurch lassen sich solche Stellenmengen binär codieren, was zu einer Reduktion der Variablenanzahl führt. Beide

Verfahren führen bei vielen Anwendungen zu einem erheblichen Speicherplatz- und Laufzeitgewinn (vgl. [Spr98]).
Bei der automatentheoretischen Betrachtungsweise der LTL spielt die Größe des aus einer LTL-Formel konstruierten Büchiautomaten eine zentrale Rolle. Der in der Arbeit vorgestellte Konstruktionsalgorithmus ist ein in vielen Werkzeugen verwendetes Verfahren. Für dieses Verfahren wurden zwei Verbesserungsansätze entwickelt, die vor bzw. nach der Büchiautomatenkonstruktion eingreifen. Vor der Büchiautomatenkonstruktion wird versucht die LTL-Formel auf syntaktischer Ebene durch eine semantisch äquivalente, aber weniger komplexe Formel zu ersetzen. Dies basiert auf einer dafür entwickelten Menge von Reduktionregeln. Die Nachbearbeitung des erzeugten Büchiautomaten setzt an einem Schwachpunkt des verwendeten Konstruktionsalgorithmus an und reduziert die Zustände eines Büchiautomaten, indem äquivalente Zustände verschmolzen werden. Das Ergebnis dieser beiden Verfahren führt zu einer teilweise erheblichen Reduktion der Büchiautomatenzustände (vgl. Abschnitt 5.5).
Die klassisch zur Überprüfung der Existenz von akzeptierenden Abläufen im Produktbüchiautomaten verwendeten Verfahren beruhen auf Einzelzustände betrachtende Tiefensuchealgorithmen. Ihr zeitlicher Aufwand ist linear zur Größe des zu untersuchenden Produktbüchiautomaten. Nach dem Auftreten eines akzeptierenden Ablaufes kann der Verifikationsprozeß mit einem Negativbescheid abgebrochen werden. Kombiniert man daher diese Verfahren mit einer on-the-fly Konstruktion des Produktbüchiautomaten erhält man ein on-the-fly Verifikationsverfahren, welches möglicherweise ohne die komplette Konstruktion des Produktbüchiautomaten auskommt. Dies ist besonders am Anfang der Entwicklung eines Systems von Bedeutung, wenn man von vielen Fehlern im System ausgehen muß.
Möchte man die Vorteile der klassischen Entscheidungsverfahren auf eine symbolische Behandlung von Zuständen übertragen, benötigt man zuerst eine on-the-fly Konstruktion des Produktbüchiautomaten, welche sich mit symbolischen Mitteln ausdrücken läßt. Daher wurde auf die Petrinetzklasse der Büchinetze zurückgegriffen. Büchinetze heben den Büchiautomatenformalismus auf Petrinetzebene an. Der Produktbüchiautomat ergibt sich dabei als Erreichbarkeitsgraph eines entsprechenden Büchinetzes. Die Handhabung von Büchinetzen mit ROBDDs läßt sich direkt von den schon betrachteten Stellen/Transitions-Netzen übernehmen.
Durch die Verschmelzung des Systempetrinetzes und des Formelbüchiautomaten zu einem Büchinetz ist es möglich, die für Petrinetze entwickelten Verfahren, z. B. die kompakte Codierung von Zuständen, auf das Produkt der Komponenten anzuwenden. Die Unterscheidung zwischen System und Spezifikation und deren separate Behandlung entfällt damit.
Mit diesem Ansatz reduziert sich das Modelcheckingproblem auf die Suche nach akzeptierenden Zuständen im Erreichbarkeitsgraphen des entsprechenden Büchinetzes, die sich selbst wieder erreichen können.
Zur Lösung dieses Problems wurden zwei Verfahren entwickelt. Das erste Verfahren

basiert auf zwei ineinander geschachtelten Fixpunktiterationen, welche die Nachfolgerfunktion von Zuständen benützen. Es handelt sich dabei um eine Modifikation des bekannten Emerson-Lei-Verfahrens. Das in dieser Arbeit neu eingeführte Verfahren betrachtet nur erreichbare Zustände und umgeht damit das Hauptproblem des Emerson-Lei-Verfahrens, welches durch die rückwärts gerichtete Fixpunktiteration auch nicht erreichbare Markierungen betrachtet. Des weiteren beantwortet das Verfahren die lange offene Frage, ob sich mit einem, nur auf der Nachfolgerfunktion beruhenden Fixpunktkalküls das LTL-Modelcheckingproblem ausdrücken läßt, positiv [HKQ98].
Mit dem neuen Verfahren lassen sich nun zwar die bis dato in symbolischen LTL-Modelcheckern eingesetzten Verfahren verbessern, doch werden die Vorteile der klassischen, auf der Betrachtung von Einzelzuständen beruhenden Verfahren nicht erreicht. Eine on-the-fly Verifikation ist mit diesem Verfahren nicht möglich, da erst nach dem Terminieren des Verfahrens über die Gültigkeit bzw. Ungültigkeit einer Spezifikation entschieden werden kann. Des weiteren ist sein Aufwand quadratisch bezüglich der Anzahl erreichbarer Zustände des Büchinetzes.
Aus diesem Grund wurde das symbolische Tiefensuche genannte Verfahren entwickelt. Das Verfahren erzeugt on-the-fly einen Mengengraphen, d. h. eine Art Erreichbarkeitsgraph dessen Knoten Zustandsmengen repräsentieren, und überprüft diesen mit Hilfe eines Tiefendurchlaufs auf akzeptierende Kreise. Die Knotenanzahl eines Mengengraphen kann dabei maximal doppelt so groß werden wie die Anzahl der erreichbaren Zustände.
Das entstandene LTL-Modelcheckingverfahren ist das erste Verfahren, welches eine symbolische Repräsentierung der Zustände mit einem on-the-fly Verifikationsverfahren verbindet und dabei eine zeitliche Komplexität der klassischen LTL-Modelcheckingverfahren erreicht.
Die Relevanz eines solchen Verfahrens läßt sich nur durch Anwendung auf in der Praxis vorkommende Problemstellungen überprüfen. Daher wurden die im Anhang A aufgeführten Beispiele bewußt unter diesem Aspekt ausgewählt. Die sich ergebenden Resultate sind sehr vielversprechend, so daß der Einsatz des Verfahrens zumindest für einen Teil der in der Praxis vorkommenden Systeme lohnend erscheint.

7.2 Ausblick

Jeder der betrachteten drei Bereiche bietet naheliegende Erweiterungsmöglichkeiten.

1. Die Beschränkung auf sichere Stellen/Transitions-Netze wurde aus Einfachheitsgründen gewählt, so daß eine Übertragung auf allgemein beschränkte Petrinetze keine Schwierigkeiten bereiten sollte. In diesem Zuge bietet sich auch der Übergang auf eine Klasse von Entscheidungsgraphen an, die nicht auf binäre Entscheidungen begrenzt sind, wie zum Beispiel die natürlichen Entscheidungsgraphen (kurz NDDs) [LR95].

2. Wie wir gesehen haben, lassen sich mit der LTL nicht alle für uns relevante Eigenschaften ausdrücken (vgl. Kapitel 5). Daher wäre eine mächtigere Spezifikationssprache wünschenswert. Anbieten würde sich die temporale Logik CTL* [EH86], die LTL enthält. Ein denkbarer Weg für einen Übergang auf diese temporale Logik ist in der Arbeit [BCG95b] dargestellt. In ihr wird ein CTL*-Modelcheckigverfahren vorgestellt, welches im Kern auf einem LTL-Modelcheckingverfahren basiert.

3. Dem Problem der Zustandsraumexplosion läßt sich nicht nur mit binären Entscheidungsgraphen beikommen. Es existieren auch Verfahren, die z. B. auf der Basis von Symmetrien des Modells [HJJJ85, Sta90] oder durch Betrachtung von partiellen Ordnungen von Zuständen (z. B. sture Mengen [Val91]) versuchen, die Erzeugung des vollständigen Zustandsraumes zu umgehen. Es wäre daher interessant zu untersuchen, inwieweit sich solche Verfahren effizient mit symbolischen Verfahren kombinieren lassen. Ein möglicher Ausgangspunkt könnte dafür die Arbeit [Tiu94] sein.

A Vergleich von Modelcheckingverfahren

Die in Kapitel 6 eingeführten LTL-Modelcheckingverfahren (BwdCycleCheck, FwdCycleCheck und SymbolicDFS) wurden prototypisch unter Verwendung der ZBDD-Bibliothek von A. Noack [Noa99] implementiert.
Zur Einordnung dieser Verfahren wurden ihnen zwei in der Praxis weit verbreitete LTL-Modelchecker gegenübergestellt.
Hierbei vertritt PROD (Version 3.1) [VHHP95] ein Prädikat/Transitions-Netz-Erreichbarkeitsanalysewerkzeug, welches auch die Verifikation von LTL-Formeln ermöglicht. PROD repräsentiert die Klasse der auf der Betrachtung von Einzelzuständen beruhenden on-the-fly Tiefensucheverfahren.
Die Klasse der symbolischen, auf dem Emerson-Lei-Verfahren aufsetzenden Verfahren wird durch NuSMV (Version 2.0.2) [CCGR99] vertreten. NuSMV ist eine Neuimplementierung des SMV-Modelcheckers [McM93], welche unter anderem auch mit dem in der Arbeit [CGH94] vorgestellten LTL-Modelcheckingverfahren für SMV erweitert wurde. Durch die Verwendung einer effizienteren ROBDD-Implementierung stellt NuSMV auch bzgl. der Performanz eine Verbesserung zu SMV dar.
Als Vergleichsobjekte wurden die beiden folgenden praxisorientierten Beispiele herangezogen.

A.1 Vergleichsobjekte

Produktionsanlage

Das erste Vergleichsobjekt stellt eine Produktionsanlage zur Verarbeitung von Metallstücken (vgl. [HD95, LL95, HDS99]) dar. Die Produnktionsanlage besteht aus zwei Transportbändern, einem Kran, einem Hubdrehtisch, einer Presse zur Verformung der Werkstücke und aus zwei Roboterarmen, die für den An- und Abtransport der Werkstücke zuständig sind.
In dem technischen Bericht [HD95] wurden mehrere Varianten der Kontrollsoftware der Produktionsanlage als sichere Stellen/Transitions-Netze modelliert. Diese Varianten unterscheiden sich einerseits durch das Abstraktionsniveau der Modellierung (Kooperationsmodell/Kontrollmodell) und andererseits durch die Anzahl der sich gleichzeitig in der Produktionsanlage befindlichen Werkstücke (1–5).

Zur Durchführung des Vergleichs wurde das detailliertere, auch die Interaktion mit den physikalischen Geräten betrachtende Kontrollmodell in den Varianten mit einem bzw. vier Werkstücken ausgewählt (s. [Hei01]). Die Variante mit einem Werkstück besitzt die kleinste und die Variante mit vier Werkstücken die größte Zustandsmenge. Sie repräsentieren somit die Eckpfeiler der Komplexität des Kontrollmodells.
Neben der Modellierung der Produktionsanlage enthält der Bericht [HD95] auch temporallogische Beschreibungen der für die Produktionsanlage zu geltenden Eigenschaften. Aus der Vielzahl der aufgeführten Eigenschaften wurden die drei folgenden Eigenschaften ausgewählt. Ihre LTL-Formeln repräsentieren die am häufigsten verwendeten Formelstrukturen.

Eigenschaft 8: „Wenn sich die Presse schließt, befinden sich beide Roboterarme außerhalb des Arbeitsbereiches der Presse."
Dies läßt sich separat für jeden Arm überprüfen, so daß die Eigenschaft durch die beiden folgenden LTL-Formeln ausdrückt werden kann:

Formel 8a: Arm 1.

$$\mathsf{G}\,((arm1\_release\_angle \wedge arm1\_release\_ext) \rightarrow (press\_stop \wedge \neg press\_at\_upper\_pos))$$

Formel 8b: Arm 2.

$$\mathsf{G}\,((arm2\_pick\_up\_angle \wedge arm2\_pick\_up\_ext) \rightarrow (press\_stop \wedge \neg press\_at\_upper\_pos))$$

Eigenschaft 31: „Solange der Kran ein Werkstück transportiert, darf der Magnet nicht deaktiviert werden."
Diese Eigenschaft läßt sich in die drei Bewegungsabläufe des Werkstückstransportes unterteilen, welche durch die folgenden drei LTL-Formeln darstellbar sind:

Formel 31a: Anheben des Magneten auf die Transporthöhe des Kranes.

$$\mathsf{G}\,((crane\_above\_deposit\_belt \wedge crane\_lift) \wedge crane\_mag\_on)\ \mathsf{U}\ (!(crane\_above\_deposit\_belt \wedge crane\_lift)))$$

Formel 31b: Bewegen des Magneten zum Zuführtransportband.

$$\mathsf{G}\,((crane\_to\_belt1 \wedge crane\_mag\_on)\ \mathsf{U}\ (!crane\_to\_belt1))$$

Formel 31c: Herunterlassen des Magneten.

$$\mathsf{G}\,((crane\_above\_feed\_belt \wedge crane\_lower) \wedge crane\_mag\_on)\ \mathsf{U}\ (!(crane\_above\_feed\_belt \wedge crane\_lower)))$$

Eigenschaft 32: „Trägt ein Roboterarm ein Werkstück, so wird sein Magnet solange nicht deaktiviert, bis sich der Arm in seiner Entladeposition befindet."
Dies drücken die beiden folgenden LTL-Formeln aus:

Formel 32a: Arm 1.

$$\mathsf{G}\,((arm1\_pick\_up\_angle \wedge arm1\_pick\_up\_ext \wedge arm1\_mag\_on) \rightarrow (arm1\_mag\_on\ \mathsf{U}\ (arm1\_release\_angle \wedge arm1\_release\_ext)))$$

Formel 32b: Arm 2.

$$\mathsf{G}\,((arm2\_pick\_up\_angle \wedge arm2\_pick\_up\_ext \wedge arm2\_mag\_on) \rightarrow (arm2\_mag\_on\ \mathsf{U}\ (arm2\_release\_angle \wedge arm2\_release\_ext)))$$

Schieberkette

Das zweite Vergleichsobjekt ist eine Kette von durch Elektromotoren gesteuerten Schiebern (vgl. [RLH96, Hei97]). Jeder Schieber der Kette kann ein Werkstück von einer Ausgangsposition in eine Endposition verschieben. Die Schieber sind so angeordnet, daß sie eine Kette bilden und die Anfangsposition eines Schiebers gerade der Endposition seines Vorgängers entspricht. Befindet sich ein Werkstück in der Anfangsposition des ersten Schiebers, so kann es durch die Kette von Schiebern zur Endposition des letzten Schiebers der Kette bewegt werden.
Die Schieberketten mit fünf bzw. neun Schiebern wurden als persistente bzw. als nicht persistente sichere Stellen/Transitions-Netze modelliert (vgl. [Hei01]). Man bezeichnet ein sicheres Stellen/Transitions-Netz als persistent, falls keine Markierung erreichbar ist, in der zwei schaltfähige Transitionen gemeinsame Stellen in ihren Vorbereichen besitzen. Für die hier betrachteten Modelcheckingverfahren ist das Schalten einer Transition vollkommen unabhängig von allen anderen ebenfalls schaltfähigen Transitionen. Insbesondere ist es nicht relevant, ob zwei schaltfähige Transitionen gemeinsame Stellen in ihren Vorbereichen haben. Für den Vergleich wurden daher willkürlich die persistenten Netze ausgewählt.
Im Bericht [Hei97] werden Eigenschaften der Schieberketten formuliert und als temporallogische Formeln angegeben. Für den Vergleich wurden exemplarisch die drei folgenden Eigenschaften ausgewählt. Sie lassen sich durch die angegebenen LTL-Formeln beschreiben. Der verwendete Parameter n repräsentiert dabei die Anzahl der Schieber in einer Schieberkette.

Eigenschaft 6: „Solange sich ein Schieber bewegt, darf kein neues Werkstück in seinem Eingangsbereich ankommen."
Dies läßt sich für jeden Schieber $i \in \{1..n\}$ einzeln durch die folgende Formel überprüfen:

Formel 6: $\mathsf{G}\,(pos_i\_full \rightarrow basic\_P_i)$

Eigenschaft 7: „Nachdem ein Schieber aktiv war, muß zuerst sein Nachfolger aktiv sein, bevor er selber wieder aktiv werden kann."
Dies läßt sich für jedes zusammenhängende Schieberpaar (i, j) mit $i \in \{1..n-1\}$ und $j = i + 1$ durch die folgende Formel überprüfen:

Formel 7:
$$\mathsf{G}\,((norm\_P_i \vee ext\_P_i) \rightarrow \mathsf{F}\,(\neg(norm\_P_i \vee ext\_P_i)\ \mathsf{U}\ (norm\_P_j \vee ext\_P_j)))$$

Eigenschaft 9: „Jedes Werkstück, welches die Schieberkette betritt, wird letztendlich die Anlage verlassen."
Diese Eigenschaft drückt die folgende Formel aus:

Formel 9:
$$\mathsf{G}\,(pos_1\_full \rightarrow \mathsf{F}\,pos_i\_full), \quad i = n + 1$$

A.2 Vergleich

Die Vergleichsmessungen wurden auf einem 800 MHz PC mit 256 MB Hauptspeicher unter dem Betriebssystem Linux durchgeführt. Als Grenzen für den Vergleich wurde ein maximaler Hauptspeicherbedarf von 256 MB, ein maximaler Plattenplatzbedarf von 512 MB und eine maximale Ausführungszeit von 12 Stunden festgelegt.
Alle Messungen, die eine dieser Grenzen überschritten haben, wurden abgebrochen und als nicht erfolgreich gewertet.
Zum Ausgleich der Schwankungsbreite der Messungen wurden für jede Formel fünf Messungen durchgeführt. Der sich daraus ergebende Mittelwert, gerundet auf halbe Sekunden, ergab den offiziellen Meßwert.
Die Teilformeln der einzelnen Eigenschaften wurden separat gemessen und anschließend zu einem Gesamtwert addiert.
Zusätzlich zu den LTL-Formeln wurden auch deren Negationen in den Vergleich mit einbezogen. Dadurch wurde eine Beurteilung der Effizienz eines on-the-fly Ansatzes ermöglicht.
Für die Verfahren aus Kapitel 6 (BwdCycleCcheck, FwdCycleCheck und SymbolicDFS) wurde eine statische Speichergröße von 32 MB zur Speicherung der ZBDD-Knoten festgelegt (vgl. [Noa99]). Dies war für eine Codierung der Petrinetzzustände der verwendeten Vergleichsobjekte ausreichend.
Des weiteren wurde für die beiden auf dem Emerson-Lei-Verfahren beruhenden Modelchecker (d. h. BwdCycleCheck und NuSMV) dem eigentlichen Modelcheckingprozeß eine Berechnung aller erreichbaren Zustände vorangestellt (vgl. Abschnitt 6.2). Ohne diesen Schritt überstieg der Hauptspeicherbedarf dieser LTL-Modelchecker nahezu immer die 256 MB Grenze.

Da der Modelchecker NuSMV als Modellierungssprache keine Petrinetze verwendet, mußten die verwendeten Petrinetzmodelle konvertiert werden. Dafür wurde das Verfahren von G. Wimmel [Wim97] eingesetzt, welches auch in dem Petrinetzwerkzeug PEP [Gra97] Verwendung findet.
Die sich für die verschiedenen Verfahren ergebenden Modelcheckingzeiten sind in den folgenden Tabellen aufgeführt:

Produktionsanlage mit einem Werkstück
($|S|$: 231, $|T|$: 202, $|\mathcal{R}_\mathcal{N}(m_0)|$: 25 000)

	Eigenschaft 8		**Eigenschaft 31**		**Eigenschaft 32**	
Verfahren	ϕ	$\neg\phi$	ϕ	$\neg\phi$	ϕ	$\neg\phi$
BwdCycleCheck	3,0 s	3,5 s	376,5 s	11,5 s	435,0 s	5,5 s
FwdCycleCheck	8,5 s	7,0 s	19,0 s	20,0 s	8,5 s	12,0 s
SymbolicDFS	5,0 s	5,5 s	9,5 s	11,0 s	12,0 s	13,0 s
PROD	138,5 s	1,0 s	1147,0 s	1,5 s	1301,5 s	278,5 s
NuSMV	-	-	-	-	-	-

Produktionsanlage mit vier Werkstücken
($|S|$: 231, $|T|$: 202, $|\mathcal{R}_\mathcal{N}(m_0)|$: 2 427 615)

	Eigenschaft 8		**Eigenschaft 31**		**Eigenschaft 32**	
Verfahren	ϕ	$\neg\phi$	ϕ	$\neg\phi$	ϕ	$\neg\phi$
BwdCycleCheck	7,5 s	34,0 s	-	29,0 s	-	76,0 s
FwdCycleCheck	521,5 s	1030,0 s	652,0 s	1090,0 s	1187,0 s	17839,5 s
SymbolicDFS	37,0 s	35,0 s	100,0 s	100,5 s	42,5 s	37,0 s
PROD	-	1,0 s	-	2,0 s	-	724,0 s
NuSMV	-	-	-	-	-	-

Schieberkette mit fünf Schiebern
($|S|$: 96, $|T|$: 89, $|\mathcal{R}_\mathcal{N}(m_0)|$: 118 624)

	Eigenschaft 6		**Eigenschaft 7**		**Eigenschaft 9**	
Verfahren	ϕ	$\neg\phi$	ϕ	$\neg\phi$	ϕ	$\neg\phi$
BwdCycleCheck	1,5 s	1,5 s	4648,0 s	8,0 s	1143,5 s	1,5 s
FwdCycleCheck	12,5 s	12,0 s	18,5 s	32,0 s	5,5 s	6,5 s
SymbolicDFS	1,5 s	1,5 s	2,0 s	2,5 s	1,0 s	1,0 s
PROD	-	1,5 s	-	598,5 s	-	5,5 s
NuSMV	6018,5 s	2748,5 s	-	-	-	-

Schieberkette mit neun Schiebern
($|S|$: 168, $|T|$: 157, $|\mathcal{R}_{\mathcal{N}}(m_0)|$: 179 774 848)

	Formel 6		Formel 7		Formel 9	
Verfahren	ϕ	$\neg\phi$	ϕ	$\neg\phi$	ϕ	$\neg\phi$
BwdCycleCheck	7,5 s	7,0 s	-	123,5 s	-	12,0 s
FwdCycleCheck	11702,5 s	9122,0 s	21338,0 s	14950,0 s	1406,0 s	2306,0 s
SymbolicDFS	38,5 s	42,5 s	41,0 s	55,0 s	18,5 s	15,0 s
PROD	-	9,0 s	-	-	-	191,5 s
NuSMV	-	-	-	-	-	-

A.3 Ergebnisse

Aus den sich ergebenden Messungen lassen sich die folgenden Schlüsse ziehen. Sie bestätigen die in Kapitel 6 geäußerten Vermutungen.

1. Mit dem LTL-Modelchecker NuSMV konnte nur in sehr wenigen Messungen ein Ergebnis erzielt werden. Da SMV sehr wohl auch bei komplexen Problemen erfolgreich eingesetzt wird, ist dies auf die unzureichende Konvertierung der Petrinetzmodelle (vgl. [Wim97]) bzw. die Ungeeigentheit der SMV-Modellierungssprache zur Beschreibung von Petrinetzen zurückzuführen. Dies belegt nochmal eindrücklich, wie wichtig ein an die Modellierungssprache und an die zu lösende Problemklasse angepaßtes Modelcheckingverfahren ist.

2. Das auf Rückwärtsiteration beruhende Emerson-Lei-Verfahren ist meist ohne eine vorherige Berechnung der Menge der erreichbaren Zustände nicht anwendbar.

3. Der Erfolg der auf Vorwärtsiteration bzw. auf Rückwärtsiteration mit vorheriger Berechnung der Menge der erreichbaren Zustände beruhenden Fixpunktverfahren ist sehr stark von der Struktur des Zustandsraumes abhängig. Vorwärts- und Rückwärtsiterationsverfahren scheinen sich dabei gut zu ergänzen.

4. Ein direkter Vergleich von on-the-fly Verfahren ist nur bedingt zulässig, da deren Erfolg stark von der Auswahl des als nächsten betrachteten Knotens abhängt. Dies ist bei den betrachteten Modelcheckern nicht steuerbar. Nichtdestotrotz zeigt sich aber in vielen Fällen der Erfolg des on-the-fly Ansatzes gegenüber Verfahren, die immer die gesammte Zustandsmenge überprüfen müssen (vgl. vor allem PROD). Bei der symbolischen Tiefensuche tritt dieser Effekt nicht so stark in Erscheinung, da die symbolische Repräsentation der Zustände dies überdeckt.

5. Symbolische Verfahren erweisen sich bei der Erzeugung einer großen Anzahl von Zuständen als vorteilhaft, da die Anzahl der betrachteten Zustände nicht zwangsläufig zu einer großen BDD-Darstellung führen. Sie ermöglichen dadurch häufig die Verifikation von Systemen, die mit der klassischen Einzelzustandserzeugung nicht mehr beherrschbar sind.

Das sich als Hauptresultat der Arbeit ergebende LTL-Modelcheckingverfahren „symbolische Tiefensuche“ erweißt sich trotz der prototypischen Implementierung als sehr effizient. Es ist gegenüber der Struktur des Zustandsraumes sehr robust und gehört bei allen Vergleichen zu den schnellsten Verfahren. Dies alles wird mit einem für gängige LTL-Modelcheckern beachtlich geringen Speicherbedarf von ca. 32 MB erreicht.
Durch eine Umsetzung der in der Arbeit vorgestellten Verbesserungen (z. B. Reduktion der LTL-Büchiautomatengröße oder Verwendung von Inhibitorkanten) hat dieses Verfahren noch einiges an Potential.

B EEMC – ein LTL-Modelchecker

Das Programm `eemc` implementiert prototypisch die drei LTL-Modelcheckingverfahren FwdCycleCheck, BwdCycleCheck und SymbolicDFS. Es besteht aus drei Teilprogrammen (`ltl2ba`, `bapn2bn` und `bneemc`), welche die drei Schritte des verwendeten LTL-Modelcheckingprozesses repräsentieren.

1. Umwandlung der LTL-Formel in einen gewöhnlichen Büchiautomaten (`ltl2ba`, vgl. Kapitel 5.3).
2. Zusammenfügen eines Büchiautomaten mit einem Petrinetz zu einem Büchinetz (`bapn2bn`, vgl. Kapitel 6.1).
3. Überprüfung der LTL-Eigenschaft anhand des Büchinetzes. Dazu kann zwischen den drei Verfahren FwdCycleCheck, BwdCycleCheck und SymbolicDFS gewählt werden (`bneemc`, vgl. Kapitel 6.2 und 6.3).

Zur Repräsentation der Büchinetzzustände werden ZBDDs verwendet (vgl. [Noa99]). Der Modelchecker kann durch das Kommando

```
eemc [-f|-b|-d] Petrinetz-Datei LTL-Formel-Datei
```

aufgerufen werden.
Dabei stehen die drei Optionen für die möglichen Modelcheckingverfahren:

- `-f` FwdCycleCheck
- `-b` BwdCycleCheck
- `-d` SymbolicDFS

Damit das Hauptprogramm `eemc` alle Teilprogramme findet, muß vorher die Umgebungsvariable `EEMC_HOME` auf das Wurzelverzeichnis der `eemc`-Distribution gesetzt werden. In Abhängigkeit des verwendeten Kommandozeileninterpreters erreicht man dies durch:

- Bourne-Shell (`/bin/sh`):
  `export EEMC_HOME="/home/dssztool/EEMC"`

- C-Shell (`/bin/sh`):
  `setenv EEMC_HOME "/home/dssztool/EEMC"`

Dabei ist das anzugebende Verzeichnis entsprechend zu ersetzen.
Der Aufruf des Programms liefert als Ergebnis die Anwort *wahr* oder *falsch*, je nachdem ob das Petrinetz die LTL-Formel erfüllt bzw. sie nicht erfüllt. Des weiteren werden einige statistische Informationen über die verwendten ZBDD-Ressourcen und die zur Verifikation benötigte Zeit ausgegeben.
Die zu verifizierenden sicheren Stellen/Transitions-Netze müssen im APNN-Format (*abstract petri net notation*, vgl. [BKK95]) vorliegen und die LTL-Formeln der folgenden Grammatik genügen:

⟨*ltl_formula*⟩	::=	⟨*atomar_proposition*⟩
	|	⟨*basic_truth_value*⟩
	|	⟨*un_operator*⟩ ⟨*ltl_formula*⟩
	|	⟨*ltl_formula*⟩ ⟨*bin_operator*⟩ ⟨*ltl_formula*⟩
	|	⟨*un_temporal_operator*⟩ ⟨*ltl_formula*⟩
	|	⟨*ltl_formula*⟩ ⟨*bin_temporal_operator*⟩ ⟨*ltl_formula*⟩
	|	'`(`' ⟨*ltl_formula*⟩ '`)`'
⟨*un_temporal_operator*⟩	::=	['`X`' | '`G`' | '`F`']
⟨*bin_temporal_operator*⟩	::=	['`R`' | '`U`']
⟨*un_operator*⟩	::=	['`!`' | '∼']
⟨*bin_operator*⟩	::=	['`&`' | '`&&`' | '`*`']
	|	['`|`' | '`||`' | '`+`']
	|	['`%`' | '`><`']
	|	['`=>`' | '`->`']
	|	['`<=`' | '`<-`']
	|	['`<=>`' | '`<->`']
⟨*basic_truth_value*⟩	::=	['`tt`' | '`true`' | '`TRUE`' | '`ff`' | '`false`' | '`FALSE`']

Literaturverzeichnis

[Ake78] Akers, S. B.: Binary Decision Diagrams. In: *IEEE Transactions on Computers* C-27 (1978), S. 509–516 {25, 33}

[And97] Andersen, H. R.: An Introduction to Binary Decision Diagrams / Lecture notes for 49285 Advanced Algorithms E97. Technical University of Denmark. Department of Information Technology. 1997. – Forschungsbericht. `http://www.it.dtu.dk/~hra` {26}

[ATB94] Aziz, A.; Tasiran, S.; Brayton, R. K.: BDD Variable Ordering for Interacting Finite State Machines. In: *31th ACM/IEEE Design Automation Conference (DAC´94)*, San Diego Convention Center, Juni 1994 {65}

[Büc60] Büchi, J. R.: On a Decision Method in Restricted Second Order Arithmetic. In: E. Nagel (Hrsg.): *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*. Stanford, CA: Stanford University Press, 1960, S. 1–12 {88}

[BCG95a] Bhat, G.; Cleaveland, R.; Grumberg, O.: Efficient On-the-Fly Model Checking for CTL*. In: *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*. San Diego, California: IEEE Computer Society Press, Juni 1995, S. 388–397 {104}

[BCG95b] Bhat, G.; Cleaveland, R.; Grumberg, O.: Efficient On-the-Fly Model Checking for CTL*. In: *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science*. San Diego, California, 26–29 Juni 1995, S. 388–397 {130}

[BKK95] Bause, F.; Kemper, P.; Kritzinger, P.: Abstract Petri Net Notation. In: *Petri Net Newsletter* (1995), Oktober, Nr. 49, S. 9–27 {140}

[BRB90] Brace, K. S.; Rudell, R. L.; Bryant, R. E.: Efficient Implementation of a BDD Package. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. Orlando, Florida: IEEE Computer Society Press, Juni 1990, S. 40–45 {43, 45}

[Bry86] Bryant, R. E.: Graph-Based Algorithms for Boolenan Function Manipulation. In: *IEEE Transactions on Computers* C-35 (1986), S. 677–691 {6, 25, 26, 33, 34}

[Bry92] Bryant, R. E.: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. In: *ACM Computing Surveys* 24 (1992), Nr. 3, S. 293–318 {6, 25, 26, 33, 34}

[CCGR99] Cimatti, A.; Clarke, E.; Giunchiglia, F.; Roveri, M.: NuSMV: A New Symbolic Model Verifier. In: Halbwachs, N. (Hrsg.); Peled, D. (Hrsg.): *Proceedings of the 11th International Conference on Computer-Aided Verification, (CAV´99)* Bd. 1633, Springer–Verlag, Juli 1999, S. 495–499 {131}

[CE81] Clarke, E. M.; Emerson, E. A.: Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In: Kozen, D. (Hrsg.): *Proceedings of the Workshop on Logics of Programs* Bd. 131. Yorktown Heights, New York: Springer-Verlag, Mai 1981, S. 52–71 {6}

[CGH94] Clarke, E. M.; Grumberg, O.; Hamaguchi, K.: Another Look at LTL Model Cecking. In: Dill, D. L. (Hrsg.): *Proceedings of the 6th International Conference on Computer-Aided Verification, (CAV´94)* Bd. 818. Stanford, California, USA: Springer–Verlag, Juni 1994, S. 415–427 {7, 108, 125, 131}

[Cho74] Choueka, Y.: Theories of Automata on ω-Tapes: A Simplified Approach. In: *Journal of Computer and System Sciences* 8 (1974), April, Nr. 2, S. 117–141 {88}

[CVWY91] Courcoubetis, C.; Vardi, M.; Wolper, P.; Yannakakis, M.: Memmory Efficient Algorithms for the Verifikation of Temporal Properties. In: Clarke, E. M. (Hrsg.); Kurshan, R. P. (Hrsg.): *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV´90)* Bd. 531, Springer–Verlag, Juni 1991, S. 233–242 {104}

[CW96] Clarke, E. M.; Wing, J. M.: Formal Methods: State of the Art and Future Directions. In: *ACM Computing Surveys* 28 (1996), Dezember, Nr. 4, S. 626–643 {5, 77}

[DB98] Drechsler, R.; Becker, B.: *Graphenbasierte Funktionsdarstellung*. Leitfäden der Informatik. Teubner, 1998 {26}

[DDR+99] Dong, Y.; Du, X.; Ramakrishna, Y. S.; Ramakrishnan, C. R.; Ramakrishnan, I. V.; Smolka, S. A.; Sokolsky, O.; Stark, E. W.; Warren,

D. S.: Fighting Livelock in the i-Protocl: A Comparative Study of Verification Tools. In: Cleaveland, W. R. (Hrsg.): *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS´99)* Bd. 1579. Amsterdam, The Netherlands: Springer–Verlag, März 1999, S. 74–88 {108}

[DE95] Desel, J.; Esparza, J.: *Cambridge Tracts in Theoretical Computer Science*. Bd. 40: *Free Choice Petri Nets*. Cambridge University Press, 1995 {9}

[EH86] Emerson, E. A.; Halpern, J. Y.: "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. In: *Journal of the ACM* 33 (1986), S. 151–178 {130}

[EL85a] Emerson, E. A.; Lei, C.-L.: Modalities for Model Checking: Branching Time Strikes Back. In: *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*. New Orleans, Louisiana: ACM Press, Januar 13–16, 1985. – Extended abstract, S. 84–96 {98}

[EL85b] Emerson, E. A.; Lei, C.-L.: Temporal Model Checking under Generalized Fairness Constraints. In: *Proceedings of the 18th Hawaii International Conference on System Sciences* Bd. 1. North-Holland, CA, 1985, S. 277–288 {98}

[EL86] Emerson, E. A.; Lei, C.-L.: Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In: *Proceedings of the 1st Annual Symposium on Logic in Computer Science, (LICS´86)*. Cambridge, MA, USA: IEEE Computer Society Press, Juni 1986, S. 267–278 {114}

[EM97] Esparza, J.; Melzer, S.: Model Checking LTL Using Constraint Programming. In: Balbo, G. (Hrsg.); Azema, P. (Hrsg.): *Proceedings of the 18th International Conference on Application and Theory of Petri Nets, (ICATPN´97)* Bd. 1248, Springer–Verlag, Juni 1997, S. 1–20 {108, 109, 111, 125}

[Eme90] Emerson, E. A.: Temporal and Modal Logic. In: van Leeuwen, J. (Hrsg.): *Handbook of Theoretical Computer Science* Bd. B: Formal Models and Semantics. Elsevier Science Publishers, 1990, S. 995–1072 {6, 78}

[FMK91] Fujita, M.; Matsunaga, Y.; Kakuda, T.: On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis. In: *Proceeding of the European Design Automation Conference (EDAC´91)*, IEEE Computer Society Press, Februar 1991, S. 50–54 {48}

[GH93] Godefroid, P.; Holzmann, G. J.: On the Verification of Temporal Properties. In: *Proceedings of the 13th International Conference on Protocol Specification, Testing, ans Verification, INWG/IFIP*. Liege, Belgium, Mai 1993, S. 109–124 {104}

[GPVW95] Gerth, R.; Peled, D.; Vardi, M.; Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. In: *Protocol Specification Testing and Verification*. Warsaw, Poland: Chapman & Hall, 1995, S. 3–18 {91, 101}

[Gra97] Grahlmann, B.: The PEP Tool. In: *Tool Presentations of Application and Theory of Petri Nets, (ATPN´97)*, 1997. – `http://theoretica.Informatik.Uni-Oldenburg.DE/~pep` {135}

[HD95] Heiner, M.; Deussen, P.: Petri Net Based Qualitative Analysis – A Case Study / Brandenburgische Technische Universität Cottbus. Institut für Informatik. Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit. 1995 (Reihe Informatik I-08/1995). – Forschungsbericht. `http://www-dssz.informatik.tu-cottbus.de/~wwwdssz` {84, 131, 132}

[HDS99] Heiner, M.; Deussen, P.; Spranger, J.: A Case Study in Design and Verification of Manufacturing System Control Software with Hierarchical Petri Nets. In: *Journal of Advanced Manufacturing Technology* 15 (1999), S. 139–152 {84, 131}

[Hei97] Heiner, M.: Verification and optimization of control programs by Petri nets without state explosion. In: Azema, P. (Hrsg.); Balbo, G. (Hrsg.): *Proceedings of the 2nd International Wokshop on Manufacturing and Petri Nets, (ICATPN´97)* Bd. 1248. Toulouse, France: Springer–Verlag, Juni 1997, S. 69–84 {133}

[Hei01] Heiner, M. *Petri Net Example Page*. Brandenburgische Technische Universität Cottbus. Institut für Informatik. `http://www-dssz.informatik.tu-cottbus.de/~wwwdssz/software/pn_examples.html`. 2001 {132, 133}

[HJJJ85] Huber, P.; Jensen, A.; Jepsen, L.; Jensen, K.: Towards Reachability Trees for High-Level Petri Nets, Springer–Verlag, 1985, S. 215–233 {130}

[HKQ98] Henzinger, T. A.; Kupferman, O.; Qadeer, S.: From Pre-historic to Post-modern Symbolic Model Checking. In: Hu, A. J. (Hrsg.); Vardi, M. Y. (Hrsg.): *Proceedings of the 10th International Conference on Computer-Aided Verification, (CAV´98)* Bd. 1427. Vancouver, BC, Canada: Springer–Verlag, Juni 1998, S. 195–206 {117, 126, 129}

[HKSV97] Hardin, R. H.; Kurshan, R. P.; Shukla, S. K.; Vardi, M. Y.: A New Heuristik for Bad Cycle Detection Using BDDs. In: Grumberg, O. (Hrsg.): *Proceedings of the 9th International Conference on Computer-Aided Verification, (CAV´97)* Bd. 1254. Haifa, Israel: Springer–Verlag, Juni 1997, S. 268–278 {116}

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall, 1985 (International Series in Computer Science) {5}

[Hol88] Holzmann, G. J.: An Improved Protocl Reachability Analysis Technique. In: *Software — Practice & Experience* 18 (1988), Februar, Nr. 2, S. 137–161 {104}

[HP96] Holzmann, G. J.; Peled, D.: On Nested Depth First Search. In: Grégoire, J.-C. (Hrsg.); Holzman, G. J. (Hrsg.); Peled, D. (Hrsg.): *The Spin Verification System* Bd. 32, American Mathematical Society, August 1996. – The Second Workshop on the Spin Verification System, S. 23–32 {104}

[INH96] Iwashita, H.; Nakata, T.; Hirose, F.: CTL Model Checking Based on Forward State Taversal. In: *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*. Washington: IEEE Computer Society Press, November 1996, S. 82–87 {116}

[ISY91] Ishiura, N.; Sawada, H.; Yajima, S.: Minimization of Binary Decision Diagrams Based on Exchanges of Variables. In: *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*. Santa Clara, CA: IEEE Computer Society Press, 1991, S. 472–475 {48}

[KPR95] Kesten, Y.; Pnuelli, A.; Raviv, L.: OBDD's LTL MC — Model Checking of Linear TL, Using OBDD's / Weizmann Institute of Science, Faculty of Mathematics and Computer Science. 1995 (CS95-13). – Technical Report {7}

[KPR98] Kesten, Y.; Pnueli, A.; Raviv, L.: Algorithmic Verification on Linear Temporal Logic Specifications. In: Larsen, K. G. (Hrsg.); Skyum, S. (Hrsg.); Winskel, G. (Hrsg.): *Proceedings of the 25th International Colloquium on Automata, Languages and Programming, (ICALP´98)* Bd. 1443. Aalborg, Denmark: Springer–Verlag, Juli 1998, S. 1–16 {126}

[Lam77] Lamport, L.: Proving the Correctness of Multiprocess Programs. In: *IEEE Transactions on Software Engineering* 3 (1977), Nr. 2, S. 125–143 {77}

[Lee59] Lee, C. Y.: Representation of Switching Circuits by Binary Decision Programs. In: *Bell System Technical Journal* 38 (1959), S. 985–999 {25, 33}

[LL92] Liaw, H.-T.; Lin, C.-S.: On the OBDD Representation of General Boolean Functions. In: *IEEE Transactions on Computers* 41 (1992), S. 661–664 {48}

[LL95] Lewerentz, C.; Lindner, T.: *Lecture Notes in Computer Science*. Bd. 891: *Formal Development of Reactive Systems: Case Study Production Cell*. New York, NY, USA: Springer–Verlag, 1995 {84, 131}

[LP85] Lichtenstein, O.; Pnueli, A.: Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In: *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*. New York: ACM Press, Januar 1985, S. 97–107 {78, 99, 104, 106}

[LR95] Lautenbach, K.; Ridder, H.: A Completion of the S-invariance Technique by means of Fixed Point Algorithms / Universität Koblenz-Landau. 1995 (10/95). – Fachberichte Informatik {116, 129}

[McM93] McMillan, K. L.: *Symbolic Model Checking*. Kluwer Academic Publishers, 1993 {131}

[Mil89] Milner, R.: *Communication and Concurrency*. Prentice Hall, 1989 (International Series in Computer Science) {5}

[Min93] Minato, S.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: *Proceedings of the 30th ACM/IEEE Design Automation Conference*. Dallas, TX: ACM Press, Juni 1993, S. 272–277 {63}

[MIY90] Minato, S.; Ishiura, N.; Yajima, S.: Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. Los Alamitos, CA: IEEE Computer Society Press, Juni 1990, S. 52–57 {45}

[MP92] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer–Verlag, 1992 {77}

[MT98] Meinel, C.; Theobald, T.: *Algorithmen und Datenstrukturen im VLSI-Design*. Springer–Verlag, 1998 {26, 28, 36}

[Noa99] Noack, A.: Ein ZBDD-Paket für effizientes Model Checking von Petrinetzen / Brandenburgische Technische Universität Cottbus. Institut für Informatik. Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit. 1999. – Forschungsbericht {61, 63, 65, 66, 76, 131, 134, 139}

[PC98] Pastor, E.; Cortadella, J.: Efficient Encoding Schemes for Symbolic Analysis of Petri Nets. In: *Proceedings of the 1998 Design Automation and Test in Europe (DATE´98)*. Paris, France: IEEE Computer Society, Februar 1998, S. 790–795 {69}

[Pet62] Petri, C. A.: Kommunikation mit Automaten / Schriften des Institutes für instrumentelle Mathematik (Bonn). 1962. – Forschungsbericht {5, 9}

[Pnu77] Pnueli, A.: The Temporal Logic of Programs. In: *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS´77)*. Providence, Rhode Island: IEEE Computer Society Press, 1977, S. 46–57 {78}

[Pnu80] Pnueli, A.: The Temporal Semantics of Concurrent Programs. In: *Theoretical Computer Science* 13 (1980), Januar, Nr. 1, S. 45–60 {6, 78}

[PRCB94] Pastor, E.; Roig, O.; Cortadella, J.; Badia, R. M.: Petri Net Analysis Using Boolean Manipulation. In: Valette, R. (Hrsg.): *Proceedings of the 15th International Conference on Application and Theory of Petri Nets* Bd. 815. Zaragoza, Spain: Springer–Verlag, Juni 1994, S. 416–435 {54, 56, 60, 61, 113}

[Rei86] Reisig, W.: *Petrinetze — Eine Einführung*. Springer–Verlag, 1986 {9}

[Rid97] Ridder, H.: *Koblenzer Schriften zur Informatik*. Bd. 6: *Analyse von Petri-Netz Modellen mit Entscheidungsdiagrammen*. Fölbach, 1997 {63, 116}

[RLH96] Rausch, M.; Lüder, A.; Hanisch, H.-M.: Combined Synthesis of Locking and Sequential Controllers. In: Smedinga, R. (Hrsg.); Spathopoulos, M. P. (Hrsg.); Kozák, P. (Hrsg.): *Proceedings of the 3rd International Wokshop on Discrete Event Systems, (WODES´96)*. Edinburgh, Scotland, U K, August 1996, S. 133–138 {133}

[Rud93] Rudell, R. L.: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In: Lightner, M. (Hrsg.): *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*. Santa Clara, CA: IEEE Computer Society Press, November 1993, S. 42–47 {48}

[SC85] Sistla, A. P.; Clarke, E. M.: The Complexity of Propositional Linear Temporal Logics. In: *Journal of the ACM* 32 (1985), Juli, Nr. 3, S. 733–749 {99}

[Sch86] Schwarz, H. R.: *Numerische Mathematik*. Teubner, 1986 {22}

[Sha38] Shannon, C. E.: A Symbolic Analysis of Relay and Switching Circuits. In: *Transactions American Institute of Electrical Engineers* 57 (1938), S. 713–723 {26}

[Spr97a] Spranger, J. *BDDs – eine Methode zur Behandlung großer Zustandsräume.* BTU Cottbus, HILES-Workshop, High-Level Synthesis Algorithms, Tools and Design. Dezember 1997 {117}

[Spr97b] Spranger, J.: Symbolic Petri Net Analysis Using Polynomials. In: *Proceedings des 4th Workshops 'Algorithmen und Werkzeuge für Petrinetze' (AWPN´97)*, 1997, S. 10 {51}

[Spr98] Spranger, J.: Combining Structural Properties and Symbolic Representation for Efficient Analysis of Petri Nets. In: Burkhard, H.-D. (Hrsg.); Czaja, L. (Hrsg.); Starke, P. (Hrsg.): *Proceedings of the Workshop on Cuncurrency, Specification and Programming 1998.* Humbolt Universität Berlin: Informatik-Bericht Nr. 110, September 1998, S. 236–244 {47, 65, 66, 67, 69, 71, 75, 76, 128}

[Sta90] Starke, H. P.: *Analyse von Petri-Netz-Modellen.* Teubner, 1990 (Leitfäden und Monographien der Informatik) {9, 130}

[Sto36] Stone, M. H.: The Theory of Representations for Boolean Algebras. In: *Transactions of the American Mathematical Society* 40 (1936), S. 37–111 {26}

[SVW87] Sistla, A. P.; Vardi, M. Y.; Wolper, P.: The Complementation Problem for Büchi Automata with Applications to Temporal Logic. In: *Theoretical Computer Science* 49 (187), Nr. 2–3, S. 217–237 {88}

[SY96] Semenov, A.; Yakovlev, A.: Combining Partial Orders and Symbolic Traversal for Efficient Verification of Asynchronous Circuits. In: *Asis-Pacific Conference on Hardware Description Languages (APCHDL´96)*, 1996, S. 567–573 {47, 65, 66}

[Tar72] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. In: *SIAM Journal on Computing* 1 (1972), Juni, Nr. 2, S. 146–160 {98, 102, 125}

[TBK95] Touati, H. J.; Brayton, R. K.; Kurshan, R.: Testing Language Containtment for ω-Automata Using BDDs. In: *Information and Computation* 118 (1995), April, Nr. 1, S. 101–109 {116}

[Tho90] Thomas, W.: Automata on Infinite Objects. In: van Leeuwen, J. (Hrsg.): *Handbook of Theoretical Computer Science* Bd. B: Formal Models

and Semantics. Elsevier Science Publishers, 1990, Kapitel 4, S. 133–191 {85}

[Tiu94] Tiusanen, M.: Symbolic, Symmetry, and Stubborn Set Searches. In: Valette, R. (Hrsg.): *Proceedings of the 15th International Conference on Application and Theory of Petri Nets* Bd. 815. Zaragoza, Spain: Springer–Verlag, Juni 1994, S. 511–530 {130}

[Val91] Valmari, A.: Stubborn Sets for Reduced State Space Generation. Berlin, Germany: Springer–Verlag, 1991, S. 491–515 {130}

[Var96] Vardi, M. Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: *Logics for concurrency: structure versus automata* Bd. 1043. New York, NY, USA: Springer-Verlag, 1996, S. 238–266 {78, 95, 97}

[VHHP95] Varpaaniemi, K.; Halme, J.; Hiekkanen, K.; Pyssysalo, T.: Prod Reference Manual / Helsinki University of Technology, Digital Systems Laboratory. 1995 (Series B: Nr. 13). – Technical Report {131}

[VW86] Vardi, M. Y.; Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In: *Proceedings 1st Annual IEEE Symposium on Logic in Computer Science, (LICS´86).* Cambridge, MA, USA: IEEE Computer Society Press, Juni 1986, S. 332–344 {78, 99, 104}

[VW94] Vardi, M. Y.; Wolper, P.: Reasoning About Infinite Computations. In: *Information and Computation* 115 (1994), November, Nr. 1, S. 1–37 {95}

[Wim97] Wimmel, G.: A BDD-based Model Checker for the PEP Tool / Department of Computing Science, University of Newcastle. 1997. – Forschungsbericht {135, 136}

[YHTM96] Yoneda, T.; Hatori, H.; Takahara, A.; Minato, S.: BDDs vs. Zero-Suppressed BDDs: For CTL Symbolic Model Checking of Petri Nets. In: *Lecture Notes in Computer Science* 1166 (1996), S. 435–449 {63, 65}

Stichwortverzeichnis