# Grundlagen und Verfahren für den Mikrosystementwurf

Dipl.-Inform. Benedikt Meurer  Prof. Dr. Rainer Brück

16.10.2008

Universität Siegen
Institut für Mikrosystemtechnik
Fachbereich Informatik und Elektrotechnik

# Inhaltsverzeichnis

# 1. Einleitung

> *„Mathematics is the gate and key to the sciences... Neglect of mathematics works injury to all knowledge, since one who is ignorant of it cannot know the other sciences of the things of this world. And what is worst, those who are thus ignorant are unable to perceive their own ignorance and so do not seek a remedy.“*
>
> *(Roger Bacon)*

**TODO** Einleitende Worte...

## 1.1. Mengenlehre

In diesem Abschnitt wird eine kurze Einführung in die in diesem Dokument verwendete Mathematik gegeben, welche im Wesentlichen auf den Ausführungen in [DP88] basiert. Für den Leser sollte dieser Abschnitt lediglich zur Auffrischung bekannten Wissens dienen, entsprechend beschränken wir uns an dieser Stelle auf die interessanten Definitionen und setzen elementare Begriffe wie *Menge*, *Relation* und *Abbildung* bzw. *Funktion* als bekannt voraus.

**Definition 1.1 (Potenzmenge)** *Sei $A$ eine beliebige Menge. Die Potenzmenge $\wp(A)$ ist die Menge, für die gilt:*

1. $\forall B \subseteq A : B \in \wp(A)$
2. $\forall B \in \wp(A) : B \subseteq A$

**Definition 1.2 (Eigenschaften von Relationen)** *Sei $M$ eine beliebige Menge und $R \subseteq M \times M$ eine binäre Relation in $M$.*

1. *$R$ heißt* reflexiv*, wenn $(x, x) \in R$ für alle $x \in M$ gilt.*
2. *$R$ heißt* transitiv*, wenn für alle $x, y, z \in M$ aus $(x, y) \in R$ und $(y, z) \in R$ folgt, dass auch $(x, z) \in R$ gilt.*
3. *$R$ heißt* symmetrisch*, wenn für alle $x, y \in M$ aus $(x, y) \in R$ folgt, dass auch $(y, x) \in R$ gilt.*

4. *$R$ heißt* antisymmetrisch*, wenn für alle $x, y \in M$ aus $(x, y) \in R$ und $(y, x) \in R$ folgt, dass $y = x$ gilt.*

**Definition 1.3 (Ordnungsrelationen)** *Sei $D$ eine beliebige Menge und $\sqsubseteq \subseteq D \times D$ eine binäre Relation in $D$.*

1. *$(D, \sqsubseteq)$ heißt* Quasiordnung*, wenn $\sqsubseteq$ reflexiv und transitiv ist.*
2. *Eine Quasiordnung $(D, \sqsubseteq)$ heißt* partielle Ordnung *(engl.:* partial order*, kurz* po*)* *oder* Halbordnung*, wenn $\sqsubseteq$ zusätzlich antisymmetrisch ist.*
3. *Eine partielle Ordnung $(D, \sqsubseteq)$ heißt* total*, wenn $x \sqsubseteq y$ oder $y \sqsubseteq x$ für alle $x, y \in D$ gilt.*
4. *Eine Quasiordnung $(D, \sqsubseteq)$ heißt* Äquivalenzrelation*, wenn $\sqsubseteq$ zusätzlich symmetrisch ist.*

Wenn aus dem Kontext ersichtlich ist, auf welche Menge Bezug genommen wird, lässt man auch die Menge weg und schreibt nur die Relation, also $\sqsubseteq$ statt $(D, \sqsubseteq)$. Entsprechend schreibt man auch lediglich $D$, wenn klar ist, wie die Menge geordnet wird.

## 1.2. Aussagenlogik

Da im weiteren Verlauf des Dokuments immer wieder auf bestimmte aussagenlogische Zusammenhänge – meist implizit, ohne dies extra zu erwähnen – zurückgegriffen wird, wiederholt dieser Abschnitt kurz die wesentlichen Definitionen und Sätze der Aussagenlogik. Wir orientieren uns dabei an der für Lehrverstaltungen zum Thema Logik üblichen Vorgehensweise. Insbesondere basiert dieser Abschnitt auf den Ausführungen in [Spr91].

In der Aussagenlogik beschäftigt man sich damit, einfache Verknüpfungen – wie „und“, „oder“ und „nicht“ – zwischen atomaren sprachlichen Gebilden zu untersuchen. Solche atomaren Gebilde können sein:

$$\begin{aligned} A &= \text{„Siegen ist eine Stadt“} \\ B &= \text{„Elefanten laufen auf dem Rüssel“} \end{aligned}$$

Diese atomaren Bestandteile können wahr oder falsch sein; für das Beispiel wissen wir, dass $A$ wahr ist, da Siegen Stadtrecht besitzt, und $B$ nach aktuellem Stand der naturwissenschaftlichen Beobachtung von Tieren falsch ist. Gegenstand der Aussagenlogik ist nun, festzustellen, auf welche Weise sich die „Wahrheitswerte“ der atomaren Bestandteile zu Wahrheitswerten von komplizierten Gebilden fortsetzen lassen, wie z.B.

$$A \text{ und } B.$$

Aus der inhaltlichen Bedeutung wissen wir, dass der Satz insgesamt falsch ist, da bereits $B$ falsch ist.

Die Aussagenlogik selbst beschäftigt sich also insbesondere nicht damit, wie sich die Wahrheitswerte für atomare Bestandteile herleiten lassen, sondern lediglich dafür, wie sich die Wahrheitswerte für komplizierte Gebilde ergeben. In diesen Untersuchungen wird letztlich ignoriert, was die zugrundeliegende inhaltliche Bedeutung der atomaren Bestandteile ist, da derartige Feinheiten für Betrachtungen irrelevant sind.

Die atomaren Bestandteile aus denen Aussagen aufgebaut werden, bezeichnet man in der Aussagenlogik als *Propositionalzeichen*. Die Menge aller Propositionalzeichen wird mit $\mathcal{AP}$ bezeichnet. Unter den Propositionalzeichen existiert ein ausgezeichnetes Element $\bot \in \mathcal{AP}$, welches als *Falsum* bezeichnet wird. Aussagen sind dann nach bestimmten grammatikalischen Regeln konstruierte Wörter über $\mathcal{AP}$, die als *Formeln* bezeichnet werden, formal:

**Definition 1.4 (Syntax der Aussagenlogik)** *Die Menge $\mathcal{L}$ aller aussagenlogischen Formeln wird wie folgt induktiv definiert:*

1. *Jedes Propositionalzeichen $p \in \mathcal{AP}$ ist eine Formel.*
2. *Sind $\phi, \psi \in \mathcal{L}$ Formeln, so sind auch $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$ und $\phi \to \psi$ Formeln.*
3. *Nur die mittels der beiden vorangegangenen Regeln gebildeten Wörter über dem Alphabet der Aussagenlogik sind Formeln.*

Bei induktiven Definitionen ist es üblich die Abgeschlossenheit der Menge implizit vorauszusetzen, statt wie im obigen Fall, die Abgeschlossenheit explizit durch den 3. Punkt der Definition zu fordern. Entsprechend werden wir in Zukunft bei Definitionen diesen Punkt nicht mehr explizit angeben.

Mit dieser Definition ist nun geregelt, wie eine syntaktisch korrekte Formel der Aussagenlogik auszusehen hat. Beispielsweise ist $p_1 \vee \neg p_2$ eine korrekte Formel, hingegen ist $p_1 \neg p_2 \vee p_3$ nicht korrekt (für $p_1, p_2, p_3 \in \mathcal{AP}$). Neben dieser Grundsyntax definiert man zusätzlich sogenannten *syntaktischen Zucker*, um die Sprache komfortabler zu gestalten. Üblich sind dabei vorallem[1]:

$$\begin{aligned} \top &:= \neg\bot \\ \phi \leftrightarrow \psi &:= (\phi \to \psi) \wedge (\psi \to \phi) \end{aligned}$$

Allerdings sind dies bisher rein syntaktische Konstrukte ohne inhaltliche Bedeutung[2]. Im Folgenden werden wir uns deshalb mit der Semantik der Aussagenlogik beschäftigen.

[1] $\top$ wird als *Versum* bezeichnet.

[2] Natürlich wird der Leser aus seiner Erfahrung heraus hier bereits eine intuitive Bedeutung zugrundgelegt haben.

## 1.2.1. Wahrheitsfunktionen

Wie zuvor erwähnt befasst sich die Aussagenlogik mit der Herleitung von Wahrheitswerten für komplizierte Gebilde – den aussagenlogischen Formeln – basierend auf Wahrheitswerten von atomaren Bestandteilen – den Propositionalzeichen. Wie sich die Wahrheitswerte der Propositionalzeichen ergeben ist dabei nicht von Interesse. Das Interesse gilt der Frage, wie sich die Wahrheitswerte der Formeln herleiten lassen. Dazu definiert man *Wahrheitsfunktionen*, die eine Menge von Wahrheitswerten auf einen Wahrheitswert abbilden und so die formale Beschreibung jeder Aussage ermöglichen.

**Definition 1.5 (Wahrheitsfunktionen)**

1. *Die Elemente der Menge* $\{\mathbf{t}, \mathbf{f}\}$ *werden als* Wahrheitswerte *bezeichnet.*
2. *Jede n-stellige Abbildung* $H : \{\mathbf{t}, \mathbf{f}\}^n \to \{\mathbf{t}, \mathbf{f}\}$ *für* $n \geq 1$ *wird als* Wahrheitsfunktion *bezeichnet.*

Den aussagenlogischen Verknüpfungen $\neg$, $\vee$, $\wedge$ und $\to$ soll eine Bedeutung in Form von Wahrheitsfunktionen $H_\neg$, $H_\vee$, $H_\wedge$ und $H_\to$ zugeordnet werden. Aus der syntaktischen Gestalt der Verknüpfungen wissen wir, dass $H_\neg$ einstellig sein muss und $H_\vee$, $H_\wedge$ sowie $H_\to$ zweistellig sein müssen.

| $a$ | $H_\neg(a)$ |
|---|---|
| **t** | **f** |
| **f** | **t** |

| $a$ | $b$ | $H_\vee(a,b)$ | $H_\wedge(a,b)$ | $H_\to(a,b)$ |
|---|---|---|---|---|
| **t** | **t** | **t** | **t** | **t** |
| **f** | **t** | **t** | **f** | **t** |
| **t** | **f** | **t** | **f** | **f** |
| **f** | **f** | **f** | **f** | **t** |

Tabellen dieser Art heissen *Wahrheitstafeln*. $H_\neg$ heisst *Negation*, $H_\vee$ heisst *Disjunktion*, $H_\wedge$ heisst *Konjunktion* und $H_\to$ heisst *Implikation*. Daneben sind noch die beiden Funktionen $H_|$ und $H_\downarrow$ von Interesse.

| $a$ | $b$ | $H_|(a,b)$ | $H_\downarrow(a,b)$ |
|---|---|---|---|
| **t** | **t** | **f** | **f** |
| **f** | **t** | **t** | **f** |
| **t** | **f** | **t** | **f** |
| **f** | **f** | **t** | **t** |

$H_|$ heisst *Shaffersche Strichfunktion* und $H_\downarrow$ heisst *Peircesche Pfeilfunktion*. Wie der Wahrheitstafel zu entnehmen ist, ist genau dann $H_|(a,b) = \mathbf{t}$, wenn nicht sowohl $a = \mathbf{t}$ als auch $b = \mathbf{t}$. Deswegen wird $H_|$ auch *NAND-Funktion* genannt. Genauso ist genau dann $H_\downarrow(a,b) = \mathbf{t}$, wenn weder $a = \mathbf{t}$ noch $b = \mathbf{t}$, weswegen $H_\downarrow$ auch als *NOR-Funktion* bezeichnet wird.

Da jede Funktion $H : \{\mathbf{t}, \mathbf{f}\}^n \to \{\mathbf{t}, \mathbf{f}\}$ mit $n > 0$ eine Wahrheitsfunktion ist, es $2^n$ $n$-Tupel mit den Komponenten **t** und **f** gibt und jedem solchen Tupel durch eine Wahrheitsfunktion der Wert **t** oder **f** zugewiesen wird, gilt offensichtlich:

**Lemma 1.1** *Es existieren* $2^{2^n}$ *verschiedene n-stellige Wahrheitsfunktionen.*

Das ist eine nicht zu unterschätzende Anzahl. Es stellt sich daher die Frage, ob eine kleine, überschaubare Menge solcher Funktionen existiert, so dass sich alle Wahrheitsfunktionen durch Verschachtelung dieser Funktionen erzeugen lassen.

**Definition 1.6 (Adäquate Mengen)**

1. *Sei* $k \in \mathbb{N}, n \geq 1$. *Eine n-stellige Wahrheitsfunktion H heißt* explizit definierbar *aus den Wahrheitsfunktionen* $H_1, \ldots, H_k$, *wenn H in der Form* $H(a_1, \ldots, a_n) = \ldots$ *definierbar ist, wobei die rechte Seite durch Komposition der Funktionen* $H_1, \ldots, H_k$ *entsteht und hierbei höchstens die Argumente* $a_1, \ldots, a_n$ *auftreten.*

2. *Eine Menge M von Wahrheitsfunktionen ist* adäquat, *falls sich jede Wahrheitsfunktion explizit aus Funktionen der Menge M definieren lässt.*

**Satz 1.1** *Die Menge* $\{H_\neg, H_\wedge, H_\vee\}$ *ist* adäquat.

**Satz 1.2**

1. $\{H_|\}$ *und* $\{H_\downarrow\}$ *sind* adäquat.

2. $\{H_|\}$ *und* $\{H_\downarrow\}$ *sind die* einzigen *einelementigen Mengen zweistelliger adäquater Wahrheitsfunktionen.*

Für die schaltungstechnische Realisierung von logischen Verknüpfungen ist Satz 1.2 von entscheidender Bedeutung, denn er besagt, dass sich alle Wahrheitsfunktionen durch Verschachtelung von $H_|$ oder $H_\downarrow$ realisieren lassen, und somit für eine prozesstechnische Verarbeitung von $n$ Wahrheitswerten nur eine dieser beiden Funktionen implementiert werden muss, statt aller $2^{2^n}$ möglichen Funktionen.

## 1.2.2. Semantik der Aussagenlogik

Ziel ist es nun jeder aussagenlogischen Formel einen Wahrheitswert – also $\mathbf{t}$ oder $\mathbf{f}$ – zuzuordnen. Man beachte, dass Formeln selbst reine Zeichenketten sind. Da die Bedeutung der Verknüpfungen $\neg$, $\vee$, $\wedge$, $\rightarrow$ die intendierte sein soll, also $H_\neg$, $H_\vee$, $H_\wedge$ bzw. $H_\rightarrow$, hängt der Wahrheitswert einer Formel von den Wahrheitswerten der in dieser Formel vorkommenden Propositionalzeichen ab.

**Definition 1.7 (Wahrheitsbelegung)** *Eine* Wahrheitsbelegung *ist eine totale Funktion* $\mathfrak{B} : \mathcal{AP} \rightarrow \{\mathbf{t}, \mathbf{f}\}$, *für die* $\mathfrak{B}(\bot) = \mathbf{f}$ *gilt.*

Statt Wahrheitsbelegung sagt man auch kurz *Belegung* oder *Variablenbelegung*, da die Propositionalzeichen innerhalb einer Formel auch als (Aussagen-)variablen bezeichnet werden. Offensichtlich lässt sich nun jede solche Wahrheitsbelegung in eindeutiger Weise zu einer *Wahrheitsbewertung* $\hat{\mathfrak{B}}$ der aussagenlogischen Formeln fortsetzen, und zwar wie folgt:

**Definition 1.8 (Wahrheitsbewertung)** *Sei $\mathfrak{B}$ eine beliebige Wahrheitsbelegung. Basierend auf $\mathfrak{B}$ wird eine* Wahrheitsbewertung $\hat{\mathfrak{B}} : \mathcal{L} \to \{\mathbf{t}, \mathbf{f}\}$ *definiert durch:*

1. $\hat{\mathfrak{B}}(p) = \mathfrak{B}(p)$, *falls* $p \in \mathcal{AP}$
2. $\hat{\mathfrak{B}}(\neg\phi) = H_\neg(\hat{\mathfrak{B}}(\phi))$
3. $\hat{\mathfrak{B}}(\phi \vee \psi) = H_\vee(\hat{\mathfrak{B}}(\phi), \hat{\mathfrak{B}}(\psi))$
4. $\hat{\mathfrak{B}}(\phi \wedge \psi) = H_\wedge(\hat{\mathfrak{B}}(\phi), \hat{\mathfrak{B}}(\psi))$
5. $\hat{\mathfrak{B}}(\phi \to \psi) = H_\to(\hat{\mathfrak{B}}(\phi), \hat{\mathfrak{B}}(\psi))$

Statt Wahrheitsbewertung sagt man auch *Bewertung*, *Semantik* oder *Interpretation*, und in der Literatur ist es ab diesem Punkt üblich nicht weiter zwischen einer Bewertung und der zugehörigen Belegung zu unterscheiden, d.h. man verwendet nur noch $\mathfrak{B}$ für beide Funktionen. Da dies jedoch häufig auf Kosten der Verständlichkeit geht, werden wir weiter streng formal zwischen einer Bewertung und der zugehörigen Belegung unterscheiden.

In der obigen Definition ist der Wahrheitswert $\hat{\mathfrak{B}}(\phi)$ einer Formel $\phi$ für Belegungen definiert, die allen Propositionalzeichen einen Wahrheitswert zuordnen, also auch den unendlich vielen, nicht in $\phi$ vorkommenden Zeichen. Intuitiv ist aber sofort ersichtlich, dass $\hat{\mathfrak{B}}(\phi)$ nur von den Wahrheitswerten abhängt, die den in $\phi$ vorkommenden Propositionalzeichen zugeordnet werden.

**Lemma 1.2 (Koinzidenzlemma)** *Sei $\phi \in \mathcal{L}$ eine Formel, und seien $\mathfrak{B}$ und $\mathfrak{B}'$ Wahrheitsbelegungen, die bezüglich der in $\phi$ vorkommenden Propositionalzeichen übereinstimmen. Dann gilt $\hat{\mathfrak{B}}(\phi) = \hat{\mathfrak{B}}'(\phi)$.*

**Definition 1.9 (Modell)**

1. *Ist $\mathfrak{B}$ eine Wahrheitsbelegung und $\phi \in \mathcal{L}$ eine Formel, so* gilt $\phi$ *unter der Belegung $\mathfrak{B}$, falls $\hat{\mathfrak{B}}(\phi) = \mathbf{t}$. Wir sagen in diesem Fall auch: $\mathfrak{B}$ ist* Modell *für $\phi$, bzw. $\phi$* gilt in dem Modell $\mathfrak{B}$ *und schreiben: $\mathfrak{B} \models \phi$. Falls $\hat{\mathfrak{B}}(\phi) = \mathbf{f}$ ist, so schreiben wir: $\mathfrak{B} \not\models \phi$.*

   *Ist $\Gamma \subseteq \mathcal{L}$ eine Menge von Formeln, so ist $\mathfrak{B}$* Modell *von $\Gamma$, falls jede Formel aus $\Gamma$ unter $\mathfrak{B}$ gilt. In diesem Fall schreiben wir $\mathcal{B} \models \Gamma$.*

2. $\Gamma \subseteq \mathcal{L}$ *heißt* erfüllbar, *falls $\Gamma$ ein Modell besitzt. Anderenfalls heißt $\Gamma$* unerfüllbar. *Ist $\Gamma = \{\phi\}$ einelementig, so sagen wir $\phi$ ist* erfüllbar *bzw. $\phi$ ist* unerfüllbar.

3. $\phi \in \mathcal{L}$ *heißt* allgemeingültig *oder* Tautologie, *falls $\phi$ unter jeder Belegung gilt. In diesem Fall schreiben wir: $\models \phi$. Anderenfalls schreiben wir: $\not\models \phi$.*

Statt *allgemeingültig* sagt man auch kurz *gültig*.

**Lemma 1.3 (Tautologielemma)** *Eine Formel $\phi \in \mathcal{L}$ ist genau dann eine Tautologie, wenn $\neg\phi$ unerfüllbar ist.*

**Beweis:** $\phi$ ist genau dann eine Tautologie, wenn $\phi$ unter jeder Belegung gilt. Dies ist genau dann der Fall, wenn $\neg\phi$ unter keiner Belegung gilt, was nichts anderes bedeutet, als dass $\neg\phi$ unerfüllbar ist. □

Ein Beispiel für eine Tautologie ist $\phi \vee \neg\phi$. Diese Formel bezeichnet man als *„tertium non datur“* (ein Drittes ist ausgeschlossen), denn für alle Belegungen ist entweder $\phi$ oder $\neg\phi$ gültig.

**Definition 1.10 (Logische Folgerung)** *Sei $\Gamma \subseteq \mathcal{L}$ eine Formelmenge und sei $\phi \in \mathcal{L}$ eine Formel. $\phi$ ist* logische Folgerung *von $\Gamma$, falls jedes Modell von $\Gamma$ auch Modell von $\phi$ ist. Wir schreiben in diesem Fall: $\Gamma \models \phi$.*

Ist $\psi \in \mathcal{L}$ eine weitere Formel und gilt $\Gamma \cup \{\psi\} \models \phi$, so schreiben wir stattdessen auch: $\Gamma, \psi \models \phi$. Ist andererseits $\Gamma = \{\alpha\}$ einelementig, so schreiben wir auch kurz: $\alpha \models \phi$.

**Definition 1.11 (Logische Äquivalenz)** *Seien $\phi, \psi \in \mathcal{L}$. $\phi$ und $\psi$ heissen* logisch äquivalent*, falls $\phi \models \psi$ und $\psi \models \phi$ gilt. In diesem Fall schreiben wir: $\phi \equiv \psi$.*

**Satz 1.3 (Metatheorem)** *Seien $\phi, \psi \in \mathcal{L}$ aussagenlogische Formeln. $\phi \equiv \psi$ gilt genau dann wenn $\models \phi \leftrightarrow \psi$ gilt.*

Mit Hilfe der Wahrheitstafeln lässt sich nun leicht überprüfen, dass auch die folgenden Äquivalenzen gelten:

**Lemma 1.4** *Seien $\phi, \psi, \pi \in \mathcal{L}$.*

$$
\begin{array}{rcll}
\phi \wedge \phi & \equiv & \phi & \textit{(Idempotenz)} \\
\phi \vee \phi & \equiv & \phi & \\
\phi \wedge \psi & \equiv & \psi \wedge \phi & \textit{(Kommutativität)} \\
\phi \vee \psi & \equiv & \psi \vee \phi & \\
(\phi \wedge \psi) \wedge \pi & \equiv & \phi \wedge (\psi \wedge \pi) & \textit{(Assoziativität)} \\
(\phi \vee \psi) \vee \pi & \equiv & \phi \vee (\psi \vee \pi) & \\
\phi \wedge (\phi \vee \psi) & \equiv & \phi & \textit{(Absorption)} \\
\phi \vee (\phi \wedge \psi) & \equiv & \phi & \\
\phi \wedge (\psi \vee \pi) & \equiv & (\phi \wedge \psi) \vee (\phi \wedge \pi) & \textit{(Distributivität)} \\
\phi \vee (\psi \wedge \pi) & \equiv & (\phi \vee \psi) \wedge (\phi \vee \pi) & \\
\neg\neg\phi & \equiv & \phi & \textit{(Doppelte Negation)} \\
\neg(\phi \wedge \psi) & \equiv & \neg\phi \vee \neg\psi & \textit{(De Morgansche Regeln)} \\
\neg(\phi \vee \psi) & \equiv & \neg\phi \wedge \neg\psi & \\
\phi \vee \psi & \equiv & \phi, \textit{ falls } \phi \textit{ Tautologie} & \textit{(Tautologieregeln)} \\
\phi \wedge \psi & \equiv & \psi, \textit{ falls } \phi \textit{ Tautologie} & \\
\phi \vee \psi & \equiv & \psi, \textit{ falls } \phi \textit{ unerfüllbar} & \textit{(Unerfüllbarkeitsregeln)} \\
\phi \wedge \psi & \equiv & \phi, \textit{ falls } \phi \textit{ unerfüllbar} &
\end{array}
$$

Im vorangegangenen Lemma fällt auf, dass die dort aufgeführten Äquivalenzen gültig bleiben, wenn alle Vorkommen von $\wedge$ durch $\vee$ ersetzt werden und umgekehrt. Dies ist ein allgemeines Gesetz der Aussagenlogik.

**Satz 1.4 (Dualitätsprinzip)** *Seien $\phi$ und $\psi$ Formeln, die nur die Zeichen $\neg$, $\vee$ und $\wedge$ enthalten, und gehen $\phi'$ und $\psi'$ aus $\phi$ bzw. $\psi$ hervor, indem alle Vorkommen von $\wedge$ durch $\vee$ und umgekehrt ersetzt werden. Ist dann $\phi \equiv \psi$, so auch $\phi' \equiv \psi'$.*

# 2. Boolesche Algebra und Schaltalgebra

## 2.1. Boolesche Algebren

Dieser Abschnitt wiederholt zunächst die Grundbegriffe Boolescher Algebren. Es wird davon ausgegangen, dass der Leser bereits über ein Grundverständnis der Thematik verfügt, so dass nur die wesentlichen Definitionen und Sätze angegeben werden.

Unter einer *Algebra* versteht man im Allgemeinen eine nichtleere Menge, auf der eine oder mehrere (möglicherweise partielle) Operationen definiert sind und in der bestimmte Axiome gelten. Man setzt dabei üblicherweise voraus, dass die Algebra unter den Operationen *abgeschlossen* ist, d.h. das Ergebnis einer Operation (so es denn existiert) ist wiederum Element der Algebra.

Eine spezielle Klasse von Algebren bilden dabei die *Booleschen Algebren*, benannt nach dem englischen Mathematiker und Philosophen George Boole. Dabei handelt es sich um eine Abstraktion der klassischen *Mengenalgebra*, die damit eine Teilmenge der Booleschen Algebren darstellt. Neben der Mengenalgebra ist die von dem amerikanischen Mathematiker Claude Shannon entwickelte *Schaltalgebra* der wichtigste Vertreter Boolescher Algebren, da sie die Grundlage für den modernen Schaltungsentwurf bildet.

Bei einer Booleschen Algebra geht man allgemein davon aus, dass eine nichtleere Menge von Objekten vorliegt, auf denen bestimmte Operationen definiert sind. Dabei kümmert man sich nicht um die Art der Operanden und Operationen und auch nicht um deren inhaltliche Bedeutung, sondern man nimmt lediglich an, dass bestimmte Rechenregeln gelten. Etwas formaler ergibt das folgende Definition:

**Definition 2.1 (Boolesche Algebra)** *Eine* Boolesche Algebra *ist eine Menge von Elementen $\mathcal{A}$, für die zweistellige Operationen $+, \cdot$ und eine einstellige Operation $^-$ definiert sind, und in der zwei Elemente* 0 *(bezeichnet als* Null*) und* 1 *(bezeichnet als* Eins*) ausgezeichnet sind derart, dass folgende Beziehungen für alle Elemente $a, b, c \in \mathcal{A}$ gelten:*

1. Assoziativgesetze *für $+$ und $\cdot$*

$$\begin{aligned} (a \cdot b) \cdot c &= a \cdot (b \cdot c) \\ (a + b) + c &= a + (b + c) \end{aligned}$$

2. Kommuntativgesetze *für $+$ und $\cdot$*

$$\begin{aligned} a \cdot b &= b \cdot a \\ a + b &= b + a \end{aligned}$$

*3.* Absorptionsgesetze

$$\begin{aligned} a \cdot (a + b) &= a \\ a + (a \cdot b) &= a \end{aligned}$$

*4.* Distributivgesetze

$$\begin{aligned} a \cdot (b + c) &= (a \cdot b) + (a \cdot c) \\ a + (b \cdot c) &= (a + b) \cdot (a + c) \end{aligned}$$

*5.* Neutrale Elemente

$$a + 0 = a \qquad a \cdot 1 = a$$

*6.* Eigenschaften des Komplements

$$a + \bar{a} = 1 \qquad a \cdot \bar{a} = 0$$

*7.* Eigenschaften von Null und Eins

$$a + 1 = 1 \qquad a \cdot 0 = 0$$

Diese Beziehungen werden auch als *Axiome der Booleschen Algebra* oder als die *Booleschen Postulate* bezeichnet. In einem gewissen Sinne sind in der obigen Liste zuviele Axiome aufgeführt, da man Teilsysteme der obigen Regeln angeben kann, aus denen bereits alle anderen Axiome folgen. Ein solches minimales System ist jedoch für die Betrachtungen in diesem Dokument irrelevant. Hier interessiert man sich lediglich dafür, dass alle Rechenregeln gelten.

**Beispiel 2.1** *Wie bereits angedeutet, ist die Potenzmenge* $\wp(M)$ *einer Menge* $M$ *mit den Operationen Durchschnitt* $\cup = +$*, Vereinigung* $\cap = \cdot$ *und Komplement, sowie den ausgezeichneten Elementen* $\emptyset = 0$ *und* $M = 1$ *eine Boolesche Algebra im Sinne von Definition 2.1. Denn wie man sich leicht überzeugen kann, gelten die Booleschen Postulate für die Mengenalgebra.*

Den angeführten Axiomen für Boolesche Algebren kommt eine wichtige Eigenschaft zu: Ersetzt man in einem Axiom gleichzeitig $+$ durch $\cdot$, $\cdot$ durch $+$, 0 durch 1 und 1 durch 0, so erhält man das andere Axiom dieser Gruppe. Man sagt auch: Die Axiome einer Gruppe sind *dual* zueinander und man nennt die angegebenen Vertauschungen *Dualisierungen*.

**Satz 2.1 (Dualitätsprinzip)** *Zu jeder Aussage, die auf die Booleschen Postulate zurückgeführt werden kann, existiert eine duale Aussage, die durch gleichzeitiges Vertauschen der Operationen* $+$ *und* $\cdot$ *sowie der neutralen Elemente* 0 *und* 1 *entsteht.*

Jede Eigenschaft Boolescher Algebren lässt sich aus den Axiomen ableiten und in jeder Aussage über Boolesche Algebren kann man die genannten Vertauschungen durchführen, also die *duale Aussage* bilden, oder anders gesprochen, die Aussage *dualisieren*. Gemäß des in Satz 2.1 formulierten *Dualitätsprinzips* gilt dann für Boolesche Algebren: Mit jeder bewiesenen Aussage ist gleichzeitig auch die dazu duale Aussage bewiesen.

Basierend auf dieser Erkenntnis lassen sich nun Folgerungen aus den Booleschen Postulaten, die nachfolgend als Lemmata angegeben sind, auf einfache Art allgemein für alle Booleschen Algebren beweisen. Die Beweise der Lemmata sind trivial und werden dementsprechend nicht angegeben.

Sei dazu im Folgenden $\mathcal{A}$ eine beliebige Boolesche Algebra.

**Lemma 2.1 (Idempotenz)** *Für alle $a \in \mathcal{A}$ gilt: $(a + a) = a = (a \cdot a)$.*

**Lemma 2.2 (Eindeutigkeit des Komplements)** *Seien $a, c \in \mathcal{A}$. Das Komplement $\bar{a}$ von $a$ ist durch die angegebenen Eigenschaften eindeutig bestimmt, d.h. gilt $a \cdot c = 0$ und $a + c = 1$, so ist $c = \bar{a}$.*

**Lemma 2.3 (Eindeutigkeit von Null und Eins)** *Seien $a, e \in \mathcal{A}$. Dann gilt:*

1. *Wenn $a + e = a$, so ist $e = 0$.*
2. *Wenn $a \cdot e = a$, so ist $e = 1$.*

**Lemma 2.4 (Involution)** *Für alle $a \in \mathcal{A}$ gilt: $\overline{(\overline{a})} = a$.*

Zu den wichtigstens abgeleiteten Regeln für Boolesche Algebren zählen die *De Morgan'schen Gesetze*, benannt nach dem englischen Mathematiker Augustus De Morgan, der auch häufig als Entdecker dieser Gesetze genannt wird, obwohl diese bereits in mitteralterlichen Schriften vor seiner Zeit Erwähnung fanden. Die Gesetze besagen, dass jede Operation + durch eine Operation $\cdot$ ausgedrückt werden kann, und umgekehrt.

**Satz 2.2 (De Morgan'sche Gesetze)** *Seien $a, b \in \mathcal{A}$. Dann gilt:*

$$\begin{aligned} \overline{(a + b)} &= \overline{a} \cdot \overline{b} \\ \overline{(a \cdot b)} &= \overline{a} + \overline{b} \end{aligned}$$

Die De Morgan'schen Gesetze finden vorallem Anwendung in mathematischen Beweisen und bei der Optimierung von Schaltungsentwürfen.

**Lemma 2.5 (Consensus)** *Sei $a, b, c \in \mathcal{A}$. Dann gilt:*

$$\begin{aligned} (a \cdot b) + (\overline{a} \cdot c) &= (a \cdot b) + (\overline{a} \cdot c) + (b \cdot c) \\ (a + b) \cdot (\overline{a} + c) &= (a + b) \cdot (\overline{a} + c) \cdot (b + c) \end{aligned}$$

**Lemma 2.6 (Simple Consensus)** *Sei $a, b \in \mathcal{A}$. Dann gilt:*

$$\begin{aligned} (a \cdot b) + (a \cdot \overline{b}) &= a \\ (a + b) \cdot (a + \overline{b}) &= a \end{aligned}$$

Der *Simple Consensus* bildet unter anderem die Grundlage für das Verfahren von Quine und McCluskey zur Minimierung von kombinatorischen Schaltungen.

### 2.1.1. Boolesche Verbände

Dieser Abschnitt betrachtet Boolesche Algebren vor einem ordnungstheoretischen Hintergrund. Dazu wird zunächst allgemein für Boolesche Algebren eine partielle Ordnung $\leq$ definiert.

**Definition 2.2** *Sei $\mathcal{A}$ eine beliebige Boolesche Algebra. Auf $\mathcal{A}$ wird eine Ordnungsrelation $\leq \subseteq \mathcal{A} \times \mathcal{A}$ wie folgt definiert:*

$$a \leq b \quad :\Leftrightarrow \quad a = a \cdot b$$

Es bleibt zu zeigen, dass es sich bei $(\mathcal{A}, \leq)$ um eine partielle Ordnung handelt.

**Lemma 2.7** *Sei $\mathcal{A}$ eine Boolesche Algebra und $\leq$ wie in Definition 2.2. Dann ist $(\mathcal{A}, \leq)$ eine partielle Ordnung.*

**Beweis:** Es ist zu zeigen, dass $\leq$ reflexiv, transitiv und antisymmetrisch ist. Dies lässt sich einzeln leicht mit Hilfe der zuvor aufgezeigten Rechenregeln für Boolesche Algebren zeigen. Seien dazu $a, b, c \in \mathcal{A}$.

1. Es gilt $a = a \cdot a$, also insbesondere $a \leq a$.
2. Nach Definition bedeutet $a \leq b$ und $b \leq c$, dass $a = a \cdot b$ und $b = b \cdot c$ gilt. Daraus folgt unmittelbar $a = a \cdot b = a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot c$, also $a \leq c$.
3. Aus $a \leq b$ und $b \leq a$ folgt $a = a \cdot b = b \cdot a = b$. □

**Satz 2.3** *Sei $\mathcal{A}$ eine Boolesche Algebra und $\leq$ wie in Definition 2.2. Dann ist $(\mathcal{A}, \leq)$ ein Verband mit kleinsten Element $\bot = 0$, größtem Element $\top = 1$, sowie dem Supremum $a \sqcup b = a + b$ und dem Infimum $a \sqcap b = a \cdot b$ für beliebige $a, b \in \mathcal{A}$.*

Da diese Eigenschaft für alle Booleschen Algebren gilt, und die Ordnungsrelation $\leq$ stets auf die gleiche Weise definiert ist, bezeichnet man Boolesche Algebren häufig auch als *Boolesche Verbände*.

**Beispiel 2.2 (Mengenalgebra)** *Sei $M$ eine beliebige Menge, und sei $\wp(M)$ eine Boolesche Algebra, in der für alle $A, B \in \wp(M)$ gilt:*

1. $A + B = A \cup B := \{x \in M \mid x \in A \vee x \in B\}$
2. $A \cdot B = A \cap B := \{x \in M \mid x \in A \wedge x \in B\}$
3. $\bar{A} = M \setminus A := \{x \in M \mid x \notin A\}$

*Dann ist $(\wp(M), \subseteq)$ ein Boolescher Verband mit kleinstem Element $\bot = \emptyset$, größtem Element $\top = M$, Supremum $A \sqcup B = A \cup B$ und Infimum $A \sqcap B = A \cap B$. Diese spezielle Boolesche Algebra wird allgemein als* Mengenalgebra *bezeichnet.*

**Definition 2.3 (Atome)** *Sei $(\mathcal{A}, \leq)$ ein Boolescher Verband. Ein Element $a \in \mathcal{A}$ heisst* Atom*, wenn $a \neq 0$ und für alle $b \in \mathcal{A}$ gilt $a \cdot b = a$ oder $a \cdot b = 0$.*

Atome sind demzufolge minimale Elemente in der eingeschränkten partiellen Ordnung $\mathcal{A} \setminus \{0\}$. Oder anders ausgedrückt, für Atome existieren neben 0 keine weiteren Elemente, welche kleiner sind als das Atom selbst. Da Atome später eine wichtige Rolle spielen werden, werden nachfolgend einige wichtige Eigenschaften von Atomen in Booleschen Verbänden aufgezeigt.

**Lemma 2.8 (Eigenschaften von Atomen)** *Sei $(\mathcal{A}, \leq)$ ein Boolescher Verband.*

1. *$a \in \mathcal{A} \setminus \{0\}$ ist Atom, wenn $b \leq a \Rightarrow b = 0$ oder $b = a$ für alle $b \in \mathcal{A}$ gilt.*
2. *Für zwei Atom $a_1, a_2 \in \mathcal{A} \setminus \{0\}$ gilt $a_1 \neq a_2 \Leftrightarrow a_1 \cdot a_2 = 0$.*
3. *Für $a \in \mathcal{A} \setminus \{0\}$ Atom und $b \in \mathcal{A}$ beliebig gilt entweder $a \leq b$ oder $a \leq \overline{b}$, aber niemals beides.*
4. *Sei $A(b) = \{a \in \mathcal{A} \,|\, a \leq b, a \text{ Atom}\}$. Dann gilt $b = \sum_{a \in A(b)} a$ für alle $b \in \mathcal{A}$.*

## 2.2. Schaltalgebra

Eine speziell für den Schaltungsentwurf interessante Boolesche Algebra stellt die 1937 von Claude Shannon in seiner Abschlussarbeit *A Symbolic Analysis of Relay and Switching Circuits* vorgestellte *Schaltalgebra* dar. Sie dient seitdem zur Beschreibung und Berechnung von *Schaltnetzen*, den kombinatorischen Schaltungen, und *Schaltwerken*, den sequentiellen Schaltungen. Während es unter Ingenieuren mittlerweile üblich ist, Schaltalgebra und Boolesche Algebra gleichzusetzen und die Begriffe beliebig zu vermischen, werden wir mit Rücksicht auf das bessere Verständnis und die formale Korrektheit, streng zwischen den beiden Begriffen – und ihren Bedeutungen – unterscheiden. Mit anderen Worten, wir betrachten die Schaltalgebra als die spezielle Boolesche Algebra, die sie ist.

**Definition 2.4 (Schaltwerte)** *Sei $\mathbb{B} = \{0, 1\}$. Auf $\mathbb{B}$ sind die Operationen $+, \cdot, \bar{}$ durch folgende Tabellen definiert:*

| $+$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| $\cdot$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| $\bar{}$ | |
|---|---|
| 0 | 1 |
| 1 | 0 |

*Die Elemente der Menge $\mathbb{B}$ bezeichnet man als* Schaltwerte.

**Definition 2.5 (Schaltfunktionen)** *Jede totale Abbildung $f : \mathbb{B}^n \to \mathbb{B}$ bezeichnet man als* Schaltfunktion. *$F_n := \{f \mid f : \mathbb{B}^n \to \mathbb{B}\}$ ist die Menge der n-stelligen Schaltfunktionen. Auf $F_n$ sind die konstanten Abbildungen $0, 1$ und die Operationen $+, \cdot, \overline{\ }$ punktweise durch*

$$\begin{array}{rcl@{\qquad}rcl} (0)(b) & := & 0 & (f+g)(b) & := & f(b) + g(b) \\ (1)(b) & := & 1 & (f \cdot g)(b) & := & f(b) \cdot g(b) \\ & & & \overline{f}(b) & := & \overline{f(b)} \end{array}$$

*definiert für alle $f, g \in F_n, b \in \mathbb{B}^n$.*

Neben der Bezeichnung Schaltfunktion findet man in der Literatur für die $f : \mathbb{B}^n \to \mathbb{B}$ auch häufig die Bezeichnung *Boolesche Funktion*, die jedoch eigentlich falsch ist, da eine Boolesche Funktion nicht auf die Trägermenge $\mathbb{B}$ beschränkt ist.

Der Leser mag sich selbst davon überzeugen, dass das nachfolgende Lemma gilt.

**Lemma 2.9**

1. *$\mathbb{B}$ ist eine Boolesche Algebra.*
2. *Für alle $n \in \mathbb{N}$ ist $F_n$ eine Boolesche Algebra.*

Wichtig hierbei ist, dass mit $\mathbb{B}$ der Bildbereich der Schaltfunktionen bereits eine Boolesche Algebra bildet. Sowohl $\mathbb{B}$ als auch die $F_n$ werden dann üblicherweise als *Schaltalgebren* bezeichnet.

**Definition 2.6 (Implikation)** *Um die partielle Ordnung $\leq$ auf $F_n$ hervorzuheben - und auch aus historischen Gründen - wird eine andere Schreib- und Sprechweise verwendet. Für $f, g \in F_n$ schreibt man $f \to g$ statt $f \leq g$ und sagt „$f$ impliziert $g$“.*

Der Hintergrund hierbei ist, dass

$$f \to g \quad \Leftrightarrow \quad \forall b \in \mathbb{B}^n : f(b) = 1 \Rightarrow g(b) = 1$$

gilt, die Relation $\leq$ also auf die prädikatenlogische Implikation zurückgeführt werden kann. Basierend auf den bisherigen Erkenntnissen lassen sich dann Rechenregeln für die Implikation herleiten, die nachfolgend als Lemma formuliert sind.

**Lemma 2.10 (Regeln für die Implikation)** *Seien $f, g, h \in F_n$ für $n \in \mathbb{N}$ beliebig, und $0, 1 \in F_n$ die entsprechenden konstanten Abbildungen aus $F_n$. Dann gilt:*

1. *$f \to f$, $0 \to f$ und $f \to 1$.*
2. *Wenn $f \to g$ und $g \to f$, dann ist $f = g$.*
3. *Wenn $f \to g$ und $g \to h$, so auch $f \to h$.*
4. *$f \to g$ gdw. $g = f + g$.*

5. $f \to g$ *und* $h \to g$ *gdw.* $f + h \to g$,
   $f \to g$ *und* $f \to h$ *gdw.* $f \to g \cdot h$.

6. $f \to g$ *gdw.* $f \cdot \overline{g} = 0$,
   $f \to g$ *gdw.* $\overline{f} + g = 1$.

7. $f = g$ *gdw.* $(\overline{f} \cdot g) + (f \cdot \overline{g}) = 0$.

8. $f \to g$ *gdw. es existiert ein* $k \in F_n$ *mit* $g = f + k$.

9. $f \cdot g \to g$ *und* $g \to f + g$.

Die Eigenschaften von Atomen aus Lemma 2.8 gelten in allen Booleschen Verbänden, und somit selbstverständlich auch in $F_n$. Daneben gilt jedoch für die speziellen Algebren $F_n$, dass sich Atome leicht identifizieren lassen.

**Lemma 2.11** *Sei* $n \in \mathbb{N}$. *Eine Schaltfunktion* $f \in F_n$ *ist genau dann ein Atom, wenn exakt ein* $b \in \mathbb{B}^n$ *existiert mit der Eigenschaft* $f(b) = 1$.

Der Leser möge sich selbst davon überzeugen, dass dieser Zusammenhang gilt.

### 2.2.1. Darstellung von Schaltfunktionen

Nachdem im vorangegangenen Abschnitt die Eigenschaften der Strukturen $F_n$ erläutert worden sind, wendet sich dieser Abschnitt dem Problem der Darstellung von Schaltfunktionen zu.

Zunächst wird dabei die wohl verbreitetste Darstellung als *Schaltformel* eingeführt. Statt Schaltformel findet man auch hier in der Literatur üblicherweise den Begriff *Boolesche Formel*, der in sofern korrekt ist, dass jede Schaltformel eine Boolesche Formel ist, aber andererseits falsch ist, da nicht jede Boolesche Formel auch eine Schaltformel ist.

**Definition 2.7 (Syntax von Schaltformeln)** *Sei* $\mathcal{V}$ *eine endliche Menge von* Schaltvariablen *und seien* $+, \cdot, \bar{}, 0, 1, (, )$ *spezielle Formelsymbole (zunächst ohne weitere Bedeutung). Die Menge der gültigen Schaltformeln* $F_\mathcal{V}$ *über* $\mathcal{V}$ *ist induktiv definiert durch:*

1. $0$ *und* $1$ *sind Schaltformeln. Jede Schaltvariable* $x \in \mathcal{V}$ *ist eine Schaltformel.*

2. *Sind* $A, B$ *Schaltformeln, so auch* $(A + B)$, $(A \cdot B)$ *und* $\overline{A}$.

3. *Nur die mit Regel 1 und 2 abgeleiteten Wörter sind gültige Schaltformeln.*

Die obige induktive Definition legt fest, wie gültige Wörter – durch welche Formeln repräsentiert werden – über einem endlichen Alphabet gebildet werden können. Es handelt sich dabei jedoch vorerst nur um syntaktische Konstrukte ohne konkrete Bedeutung.

**Definition 2.8 (Semantik von Schaltformeln)** *Sei $\mathcal{V} = \{x_1, \ldots, x_n\}$ eine Menge von Schaltvariablen, seien $A, B \in F_{\mathcal{V}}$ Schaltformeln über $\mathcal{V}$ und $b = (b_1, \ldots, b_n) \in \mathbb{B}^n$. Die Abbildung $\langle\cdot\rangle : F_{\mathcal{V}} \to F_n$ ist wie folgt induktiv über die Struktur von Schaltformeln definiert:*

$$\begin{aligned}
\langle 0\rangle &:= 0 \\
\langle 1\rangle &:= 1 \\
\langle x_i\rangle(b) &:= b_i \quad \textit{für alle } i = 1, \ldots, n \\
\langle \overline{A}\rangle &:= \overline{\langle A\rangle} \\
\langle A \cdot B\rangle(b) &:= \langle A\rangle(b) \cdot \langle B\rangle(b) \\
\langle A + B\rangle(b) &:= \langle A\rangle(b) + \langle B\rangle(b)
\end{aligned}$$

*Die so definierte Abbildung $\langle\cdot\rangle$ wird als* Schaltsemantik *bezeichnet.*

Wie erläutert sind Schaltformeln zunächst einfach Wörter über einem endlichen Alphabet, ohne konkrete Bedeutung. Erst durch die Semantikfunktion $\langle\cdot\rangle$ aus Definition 2.8 wird den Schaltformeln eine Bedeutung in Form einer Schaltfunktion zugeordnet.

**Beispiel 2.3** *Sei $\mathcal{V} = \{x_1, x_2, x_3\}$ eine Menge von Schaltvariablen und $A = x_1 \cdot x_2 + \overline{x_3}$ eine Schaltformel über $\mathcal{V}$. Durch die Semantikfunktion wird der Formel $A$ eine Schaltfunktion $f = \langle A\rangle \in F_3$ zugeordnet, die wie folgt definiert ist:*

$$f(b_1, b_2, b_3) \quad = \quad \begin{cases} 1, & \textit{falls } (b_1 = 1 \wedge b_2 = 1) \vee b_3 = 0 \\ 0, & \textit{sonst} \end{cases}$$

*Dieses Beispiel zeigt auch die enge Verschandtschaft der Schaltalgebra zur Aussagenlogik.*

Wie aus Definition 2.8 leicht ersichtlich, ist die Zuordnung von Schaltformeln zu Schaltfunktionen nicht eindeutig. In einer Schaltformel aus $F_{\{x_1,\ldots,x_n\}}$ muss nicht jedes $x_i$ vorkommen. Beispielsweise ist für $x_1 \cdot \overline{x_2} + x_3$ unklar, ob es sich um eine Schaltformel über $\mathcal{V} = \{x_1, x_2, x_3\}$ oder über $\{x_1, x_2, x_3, x_4\}$ oder über einer anderen endlichen Obermenge von $\mathcal{V}$ handelt. Für $x_1 \cdot \overline{x_2} + x_3$ ist lediglich bekannt, dass es sich um die Darstellung einer *mindestens* dreistelligen Schaltfunktion handelt.

**Konvention 2.1** *Zum einfacheren Umgang mit Schaltformeln und Schaltfunktionen werden folgende Vereinbarungen getroffen:*

1. *Für Formeln wie für Ausdrücke in Funktionen bindet das Zeichen $\cdot$ stärker als das Zeichen $+$, und das Zeichen $\cdot$ muss nicht notiert werden. Das bedeutet $x_1x_2 + x_3$ steht für $(x_1 \cdot x_2) + x_3$.*

2. *Wegen der Kommutativität gilt $f_1 \cdot f_2 = f_2 \cdot f_1$. Deshalb wird auch für Formel, sofern nicht anders gesagt, nicht zwischen $A_1 \cdot A_2$ und $A_2 \cdot A_1$ unterschieden. Gleiches gilt für $A_1 + A_2$ und $A_2 + A_1$.*

3. *Wegen der Assoziativität gilt* $(f_1 \cdot f_2) \cdot f_3 = f_1 \cdot (f_2 \cdot f_3)$. *Deshalb wird für beide Ausdrücke die verkürzte Schreibweise* $f_1 \cdot f_2 \cdot f_3$ *verwendet. Analog wird* $f_1 + f_2 + f_3$ *benutzt. Für Schaltformeln* $A_1, A_2, A_3$ *gilt*

$$\begin{aligned} \langle A_1 \cdot (A_2 \cdot A_3) \rangle &= \langle A_1 \rangle \cdot (\langle A_2 \rangle \cdot \langle A_3 \rangle) \\ &= (\langle A_1 \rangle \cdot \langle A_2 \rangle) \cdot \langle A_3 \rangle \\ &= \langle (A_1 \cdot A_2) \cdot A_3 \rangle \,. \end{aligned}$$

*Entsprechend wird auch hier die Schreibweise* $A_1 \cdot A_2 \cdot A_3$ *und* $A_1 + A_2 + A_3$ *verwendet bei der Darstellung von Funktionen.*

In der Literatur ist es üblich, ab diesem Zeitpunkt nicht weiter zwischen Schaltfunktionen und Schaltformeln zu unterscheiden. Dieser Ansatz kann jedoch in vielen Fällen verwirrend sein. Entsprechend wird in diesem Dokument weiterhin sorgfältig unterschieden zwischen einer Schaltfunktion und ihrer Darstellung als Schaltformel. Das bedeutet, die beiden Formeln $x_1\overline{x_2} + x_2x_3$ und $x_1\overline{x_2} + x_2x_3 + x_1x_3$ sind verschiedene Formeln in $\mathcal{V} = \{x_1, x_2, x_3\}$.

Andererseits gilt aber $\langle x_1\overline{x_2} + x_2x_3 \rangle = \langle x_1\overline{x_2} + x_2x_3 + x_1x_3 \rangle$, d.h. die Bedeutung der Schaltformeln stimmt überein. Diese semantische Übereinstimmung von Schaltformeln wird üblicherweise als *semantische Äquivalenz* oder einfach *Äquivalenz* bezeichnet.

**Definition 2.9 (Äquivalenz von Schaltformeln)** *Sei* $\mathcal{V}$ *eine endliche Menge von Schaltvariablen und seien* $A, B \in F_{\mathcal{V}}$ *Schaltformeln über* $\mathcal{V}$. *A und B heissen* semantisch äquivalent *oder kurz* äquivalent, *wenn* $\langle A \rangle = \langle B \rangle$ *gilt. In diesem Fall schreibt man* $A \equiv B$. *Gilt* $\langle A \rangle \neq \langle B \rangle$, *so schreibt man* $A \not\equiv B$.

Für das vorangegangene Beispiel schreibt man also $x_1\overline{x_2} + x_2x_3 \equiv x_1\overline{x_2} + x_2x_3 + x_1x_3$. Im Folgenden wird eine vereinfachte Schreibweise zur Notation von bedingt nicht negierten oder negierten Schaltformeln und Schaltfunktionen eingeführt, die in vielen Fällen eine einfachere Darstellung von Zusammenhängen erlaubt.

**Konvention 2.2** *Für Schaltfunktionen* $f \in F_n$ *seien die abkürzenden Schreibweisen*

$$f^0 := \overline{f} \qquad f^1 := f$$

*definiert. Analog seien für Schaltformeln* $A \in F_{\mathcal{V}}$ *die abkürzenden Schreibweisen*

$$A^0 := \overline{A} \qquad A^1 := A$$

*definiert.*

Beispielsweise steht $f^1g^0$ für die Funktion $(f \cdot \overline{g})$ und $x_1^1x_2^1x_3^0$ steht für die Schaltformel $(x_1x_2\overline{x_3})$.

**Definition 2.10 (Produktterme, Summenterme, Minterme, Maxterme)** *Sei* $\mathcal{V} = \{x_1, \ldots, x_n\}$ *und* $e_1, \ldots, e_n \in \mathbb{B}$.

1. *Formeln der Form* $x_{i_1}^{e_1} \cdot \ldots \cdot x_{i_k}^{e_k}$ *mit* $x_{i_j} \neq x_{i_k}$ *für* $j \neq k$ *bezeichnet man als* Produktterme.

2. *Formeln der Form* $x_{i_1}^{e_1} + \ldots + x_{i_k}^{e_k}$ *mit* $x_{i_j} \neq x_{i_k}$ *für* $j \neq k$ *bezeichnet man als* Summenterme.

3. *Einen Produktterm der Form* $x_1^{e_1} \cdot \ldots \cdot x_n^{e_n}$ *bezeichnet man als* Minterm.

4. *Einen Summentern der Form* $x_1^{e_1} + \ldots + x_n^{e_n}$ *bezeichnet man als* Maxterm.

Die Bezeichnung Minterm erklärt sich dadurch, dass die $\langle x_1^{e_1} \cdot \ldots \cdot x_n^{e_n} \rangle$ minimale Elemente in $F_n \setminus \{0\}$ bilden, dass also $\langle x_1^{e_1} \cdot \ldots \cdot x_n^{e_n} \rangle$ Atome sind. Entsprechend gilt offensichtlich, dass die $\langle x_1^{e_1} + \ldots + x_n^{e_n} \rangle$ maximale Elemente in $F_n \setminus \{1\}$ sind.

**Definition 2.11** *Seien* $b = (b_1, \ldots, b_n) \in \mathbb{B}^n$, $e_1, \ldots, e_k \in \mathbb{B}$ *und sei* $t = x_{i_1}^{e_1} \ldots x_{i_k}^{e_k}$ *ein Produktterm über* $\{x_1, \ldots, x_n\}$. *Dann sei*

$$\langle t = 1 \rangle (b) \quad := \quad (b'_1, \ldots, b'_n)$$

*definiert mit* $b'_{i_j} = b_{i_j}$ *für* $j \notin \{1, \ldots, k\}$ *bzw.* $b'_{i_j} = e_j$ *für* $j \in \{1, \ldots, k\}$.

Für jeden Produktterm $t$ ist $\langle t = 1 \rangle$ offensichtlich eine Abbildung von $\mathbb{B}^n$ nach $\mathbb{B}^n$. Dabei bestimmen die Exponenten von $t$, welche Stellen in $(b_1, \ldots, b_n)$ wie geändert werden. Betrachten wir dazu das folgende Beispiel.

**Beispiel 2.4** *Sei* $t = \bar{x}_1 x_3 \bar{x}_4 = x_1^0 x_3^1 x_4^0$ *ein Produktterm und sei* $b = (0, 1, 0, 1) \in \mathbb{B}^4$. *Dann ist* $\langle t = 1 \rangle (b) = (0, 1, 1, 0)$. *Der Wert von* $b_2$ *bleibt unverändert, da* $x_2$ *in* $t$ *nicht vorkommt. An den Stellen* 1, 3 *und* 4 *werden die Exponenten von* $t$ *übernommen, unabhängig davon, wie die Belegung von* $b$ *an diesen Stellen war.*

Offensichtlich existiert ein einfacher Zusammenhang zwischen den Funktionen $\langle t \rangle$ und $\langle t = 1 \rangle$, der nachfolgend als Lemma dargestellt ist.

**Lemma 2.12** *Sei* $n \geq 1$. *Dann gilt für jeden Produktterm* $t \in F_{\{x_1, \ldots, x_n\}}$ *und jedes Tupel* $b \in \mathbb{B}^n$*:*

$$\langle t = 1 \rangle (b) = b \quad \Leftrightarrow \quad \langle t \rangle (b) = 1$$

Damit sind wir nun in der Lage den Begriff der *Subfunktion* einer Schaltfunktion $f$ nach einem Produktterm $t$ einzuführen. Um diese Definition unabhängig von der konkreten Darstellung der Funktion $f$ zu machen, wurde zuvor die Abbildung $\langle t = 1 \rangle$ eingeführt.

**Definition 2.12 (Subfunktion)** *Sei* $n \in \mathbb{N}$ *beliebig. Seien weiter* $f \in F_n$ *eine Schaltfunktion und* $t \in F_{\{x_1, \ldots, x_n\}}$ *ein Produktterm. Dann ist die* Subfunktion $f_t : \mathbb{B}^n \to \mathbb{B}$ *von* $f$ *nach* $t$ *definiert durch:*

$$f_t \quad := \quad f \circ \langle t = 1 \rangle$$

Nach Definition hat $f_t$ somit die gleiche Stelligkeit wie $f$. Das Ergebnis der Subfunktion $f_t$ ist dann aber unabhängig von den durch $\langle t = 1 \rangle$ konstant abgebildeten Stellen. Vereinfacht ausgedrückt bedeutet dies, dass $f_t$ damit unabhängig ist von den in $t$ vorkommenden Variablen. Bekanntlich sagt ein Beispiel mehr als tausend Worte.

**Beispiel 2.5** *Sei $f \in F_4$ die wie folgt definierte Schaltfunktion.*

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

*Sei weiter $t = \bar{x}_1 x_3$ ein Produktterm über $\{x_1, \ldots, x_4\}$. Dann gilt beispielsweise*

$$f_t(1,1,0,0) = f(\langle t = 1 \rangle (1,1,0,0)) = f(\langle \bar{x}_1 x_3 = 1 \rangle (1,1,0,0)) = f(0,1,1,0) = 0.$$

*Insgesamt ergibt sich für $f_t$ folgende Wertetabelle:*

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f_t$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f_t$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

*Der Leser möge sich selbst davon überzeugen, dass diese Tabelle korrekt ist. Wie man leicht sieht kann es durchaus vorkommen, dass die Subfunktion auch von nicht in $t$ vorkommenden Variablen unabhängig wird, was sich in diesem Fall an $x_4$ zeigt.*

Die Berechnung der Subfunktion $f_t$ anhand von Wertetabellen ist mitunter recht mühsam. Ist eine Darstellung der Schaltfunktion $f$ als Schaltformel bekannt, so lässt sich mit Hilfe einer einfachen Umformung die Schaltformel für die Subfunktion $f_t$ herleiten.

**Definition 2.13 (Subformel)** *Sei $n \in \mathbb{N}$ und $\mathcal{V} = \{x_1, \ldots, x_n\}$. Sei dann $A \in F_\mathcal{V}$ eine beliebige Schaltformel und $t = x_{i_1}^{e_1} \ldots x_{i_k}^{e_k} \in F_\mathcal{V}$ ein Produktterm über $\mathcal{V}$. Die* Subformel *$A_t$ von $A$ nach $t$ erhält man indem jedes Vorkommen von $x_{i_j}$ in $A$ durch $e_j$ ersetzt wird.*

Der Begriff *Subformel* sollte nicht verwechselt werden mit dem Begriff der Teilformel. Eine Subformel ist i.d.R. nicht syntaktischer Bestandteil der ursprünglichen Formel. Vielmehr handelt es sich dabei um das formelmäßige Pendant zur Subfunktion, wie das nachfolgende Lemma zeigt.

**Lemma 2.13** *Sei $n \in \mathbb{N}$ und $\mathcal{V} = \{x_1, \ldots, x_n\}$. Für alle $f \in F_n$ und alle $A \in F_\mathcal{V}$ sowie alle Produktterme $t \in F_\mathcal{V}$ gilt:*

$$f = \langle A \rangle \quad \Rightarrow \quad f_t = \langle A_t \rangle$$

**Beispiel 2.6** *Greifen wir nun noch einmal die Funktion aus dem vorangegangenen Beispiel auf. Es gilt*

$$f \quad = \quad \langle x_1 + \bar{x}_2 x_3 + x_2 \bar{x}_3 x_4 \rangle \quad = \quad \langle A \rangle \, .$$

*Sei wieder $t = \bar{x}_1 x_3$. Für $A_t$ muss nun jedes Vorkommen von $x_1$ durch 0 und jedes Vorkommen von $x_3$ durch 1 ersetzt werden. Damit ergibt sich $A_t = 0 + \bar{x}_2 \cdot 1 + x_2 \cdot \bar{1} \cdot x_4$. Offensichtlich gilt $A_t \equiv \bar{x}_2$ und damit $f_t = \langle \bar{x}_2 \rangle$, was dem erwarteten Ergebnis entspricht.*

Nun ist es recht einfach, den folgenden Satz zu beweisen, der die historisch älteren berühmten Ergebnisse, wie sie anschließend als Korollare formuliert werden, als Spezialfälle enthält.

**Satz 2.4** *Sei $n \in \mathbb{N}$ und $\mathcal{V} = \{x_1, \ldots, x_n\}$. Seien weiter $t_1, \ldots, t_m \in F_\mathcal{V}$ Produktterme mit $\left\langle \sum_{i=1}^{m} t_i \right\rangle = 1$ und $f \in F_n$ eine Schaltfunktion. Dann gilt:*

1. $f = \sum_{i=1}^{m} \langle t_i \rangle \, f_{t_i}$
2. $f = \prod_{i=1}^{m} \left( \overline{\langle t_i \rangle} + f_{t_i} \right)$

**Beweis:** Für jedes $b \in \mathbb{B}^n$ existiert mindestens ein $i \in \mathbb{N}$ mit $\langle t_i \rangle (b) = 1$, denn nach Voraussetzung gilt $\left\langle \sum_{i=1}^{m} t_i \right\rangle = 1$. Folglich muss $\langle t_i = 1 \rangle (b) = b$ gelten und somit

$$f_{t_i}(b) \quad = \quad f\left(\langle t_i = 1 \rangle (b)\right) \quad = \quad f(b).$$

Wir definieren

$$I_b \quad = \quad \{i \mid \langle t_i \rangle (b) = 1\}$$

und

$$J_b \quad = \quad \{i \mid \langle t_i \rangle (b) = 0\}.$$

Damit lassen sich die beiden Behauptungen leicht beweisen:

1.

$$\begin{aligned}\left(\sum_{i=1}^{m}\langle t_i\rangle\, f_{t_i}\right)(b) &= \sum_{i\in I_b}\langle t_i\rangle\,(b)\, f_{t_i}(b) + \sum_{i\in J_b}\langle t_i\rangle\,(b)\, f_{t_i}(b)\\ &= \sum_{i\in I_b} f_{t_i}(b)\\ &= \sum_{i\in I_b} f(b)\\ &= f(b)\end{aligned}$$

2.

$$\begin{aligned}\left(\prod_{i=1}^{m}\left(\overline{\langle t_i\rangle} + f_{t_i}\right)\right)(b) &= \prod_{i\in I_b}\left(\overline{\langle t_i\rangle\,(b)} + f_{t_i}(b)\right)\cdot\prod_{i\in J_b}\left(\overline{\langle t_i\rangle\,(b)} + f_{t_i}(b)\right)\\ &= \prod_{i\in I_b} f_{t_i}(b)\\ &= \prod_{i\in I_b} f(b)\\ &= f(b)\end{aligned}$$

□

**Korollar 2.1 (Shannon-Zerlegung)** *Sei $n \in \mathbb{N}$. Für jede Funktion $f \in F_n$ und jede Variable $x \in \{x_1, \ldots, x_n\}$ gilt:*

1. $f = \langle \bar{x}\rangle\, f_{\bar{x}} + \langle x\rangle\, f_x$
2. $f = (\langle \bar{x}\rangle + f_x)\,(\langle x\rangle + f_{\bar{x}})$

**Beweis:** Trivial, da $\langle x + \bar{x}\rangle = 1$. □

**Korollar 2.2 (Mintermnormalform, Maxtermnormalform)** *Sei $n \in \mathbb{N}$, $f_n \in F_n$ eine $n$-stellige Schaltfunktion und seien $x_1, \ldots, x_n$ Schaltvariablen. Dann gilt:*

1. Mintermnormalform $f = \left\langle \sum_{b\in\mathbb{B}^n} x_1^{b_1} \ldots x_n^{b_n} \cdot f(b) \right\rangle$
2. Maxtermnormalform $f = \left\langle \prod_{b\in\mathbb{B}^n} \left(x_1^{\bar{b}_1} + \ldots + x_n^{\bar{b}_n} + f(b)\right) \right\rangle$

Das bedeutet, zu jeder Schaltformel existiert eine äquivalente Schaltformel in Mintermnormalform und eine äquivalente Formel in Maxtermnormalform. Betrachten wir dazu wieder ein Beispiel.

**Beispiel 2.7** *Die Schaltfunktion $f \in F_4$ sei durch folgende Wertetabelle definiert.*

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

*In der Darstellung als Mintermnormalform werden die Summanden, welche mit $f(b) = 0$ multipliziert werden unterdrückt. Es werden als lediglich die 1-Stellen notiert. Damit ergibt sich*

$$f = \left\langle \begin{array}{l} \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1x_2\bar{x}_3\bar{x}_4 \\ +\bar{x}_1x_2\bar{x}_3x_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1\bar{x}_2x_3x_4 + x_1x_2\bar{x}_3\bar{x}_4 \end{array} \right\rangle .$$

*Entsprechend werden in der Darstellung als Maxtermnormalform die Terme, welche zu $f(b) = 1$ addiert werden, unterdrückt und wir erhalten*

$$f = \left\langle \begin{array}{l} (x_1 + x_2 + x_3 + x_4)\ (x_1 + \bar{x}_2 + \bar{x}_3 + x_4) \\ (x_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)\ (\bar{x}_1 + x_2 + x_3 + x_4) \\ (\bar{x}_1 + x_2 + \bar{x}_3 + x_4)\ (\bar{x}_1 + \bar{x}_2 + x_3 + \bar{x}_4) \\ (\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + x_4)\ (\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \end{array} \right\rangle .$$

*Als Daumenregel kann man sich merken: Für die Mintermnormalform werden aus der Wertetabelle alle 1-Stellen nicht-negiert ausgelesen, für die Maxtermnormalform werden alle 0-Stellen negiert ausgelesen.*

Für die Repräsentation von Schaltfunktionen (z.B. in einem Programm) wurden bisher neben der Wertetabelle nur verschiedene darstellende Schaltformeln behandelt. Man kann nun noch einen Schritt weiter gehen und bestimmte Schaltformeln in einer kompakten Form kodieren. Dabei werden nur Formeln in Min- bzw. Maxtermnormalform betrachtet.

Sei dazu $f$ die Schaltfunktion aus dem vorherigen Beispiel. $f$ in Mintermnormalform ist offensichtlich festgelegt durch die Menge

$$\left\{ \begin{array}{l} \bar{x}_1\bar{x}_2\bar{x}_3x_4, \bar{x}_1\bar{x}_2x_3\bar{x}_4, \bar{x}_1\bar{x}_2x_3x_4, \bar{x}_1x_2\bar{x}_3\bar{x}_4, \\ \bar{x}_1x_2\bar{x}_3x_4, x_1\bar{x}_2\bar{x}_3x_4, x_1\bar{x}_2x_3x_4, x_1x_2\bar{x}_3\bar{x}_4 \end{array} \right\}$$

von Mintermen. Minterme wiederum werden durch die Exponenten der Variablen beschrieben. Es genügt dementsprechend, sich diese zu notieren, d.h.

$$\{0001, 0010, 0011, 0100, 0101, 1001, 1011, 1100\} .$$

Liest man die Elemente dieser Menge als Zahlen zur Basis 2, so erhält man eine Nummerierung der Minterme. Die einzelnen Minterme werden dann mit $m_i$ bezeichnet, wobei $i$ die sich aus der Binärzahl ergebende Nummer des Minterms ist. Damit ergibt sich als kompakte Notation für $f$ die Menge

$$\{m_1, m_2, m_3, m_4, m_5, m_9, m_{11}, m_{12}\}.$$

Alternativ kann natürlich auch die Maxtermnormalform nach dem gleichen Schema kodieren. Für Maxterme verwendet man dann die Bezeichnung $M_i$.

Daneben gibt es eine weitere Darstellungsmöglichkeit, welche die Struktur von Schaltfunktionen ausnutzt, ohne auf eine Darstellung durch Formeln zurückgreifen zu müssen. Ausgangspunkt ist die Shannon-Zerlegung einer Funktion (vgl. Korollar 2.1).

**Definition 2.14 (Binary Decision Diagram)** *Sei $n \in \mathbb{N}$, $f \in F_n$ und $x_1, \ldots, x_n$ eine geordnete Menge von Variablen. Der gerichtete azyklische Graph $BDD(f; x_1, \ldots, x_n)$ besteht aus der Knotenmenge $V$ und der gerichteten Kantenmenge $E \subseteq V \times V \times M$, die wie folgt definiert sind:*

1. $V = \{0,1\} \cup \left\{ f_t \,|\, t = x_{i_1}^{e_1} \ldots x_{i_k}^{e_k},\ f_t \neq 0, f_t \neq 1 \right\}$
2. $M = \{x_i^{e_i} \,|\, i = 1, \ldots, n,\ e_i = 0, 1\}$
3. $(g, h, x^e) \in E \ \Leftrightarrow\ h = g_{x^e}$

*Einen derartigen Graphen bezeichnet man als* Binary Decision Diagram (BDD) *oder* Subfunktionsgraph.

**Beispiel 2.8** *Betrachten wir beispielhaft die Funktion*

$$f \ = \ \langle \bar{x}_2 x_4 + x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 \rangle$$

*Dann ist*

$$\begin{aligned} f_{x_1} &= \langle \bar{x}_2 x_4 + x_2 \bar{x}_3 \bar{x}_4 \rangle \\ f_{\bar{x}_1} &= \langle \bar{x}_2 x_4 + x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_2 x_3 + x_2 \bar{x}_3 \rangle \\ &= \langle \bar{x}_2 x_4 + \bar{x}_2 x_3 + x_2 \bar{x}_3 \rangle \end{aligned}$$

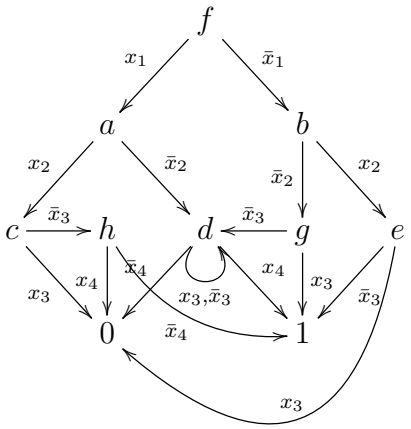
*und*

$$\begin{aligned} f_{x_1 x_2} &= \langle \bar{x}_3 \bar{x}_4 \rangle & f_{x_1 \bar{x}_2} &= \langle x_4 \rangle \\ f_{\bar{x}_1 x 2} &= \langle \bar{x}_3 \rangle & f_{\bar{x}_1 \bar{x}_2} &= \langle x_4 + x_3 \rangle \\ f_{x_1 x_2 x_3} &= 0 & f_{x_1 x_2 \bar{x}_3} &= \langle \bar{x}_4 \rangle \\ f_{x_1 \bar{x}_2 x_3} &= \langle x_4 \rangle & f_{x_1 \bar{x}_2 \bar{x}_3} &= \langle x_4 \rangle \\ f_{\bar{x}_1 x_2 x_3} &= 0 & f_{\bar{x}_1 x_2 \bar{x}_3} &= 1 \\ f_{\bar{x}_1 \bar{x}_2 x_3} &= 1 & f_{\bar{x}_1 \bar{x}_2 \bar{x}_3} &= \langle x_4 \rangle \\ f_{x_1 \bar{x}_2 x_4} &= 1 & f_{x_1 \bar{x}_2 \bar{x}_4} &= 0 \\ f_{x_1 x_2 \bar{x}_3 x_4} &= 0 & f_{x_1 x_2 \bar{x}_3 \bar{x}_4} &= 1 \end{aligned}$$

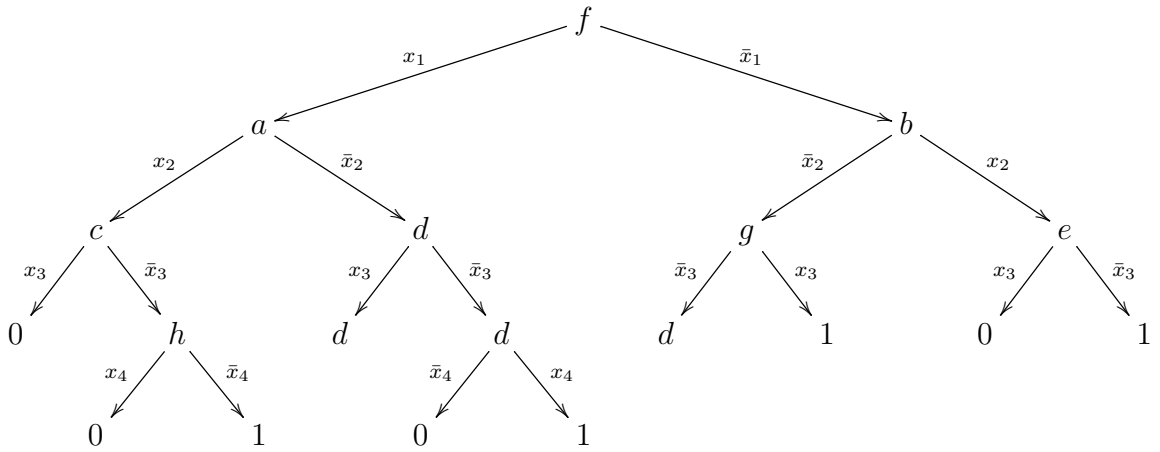
*Wegen $f_{x_1\bar{x}_2} = f_{x_1\bar{x}_2x_3} = f_{x_1\bar{x}_2\bar{x}_3}$ müssen wir keine weiteren Subfunktionen berechnen. Wir wählen die folgenden Bezeichnungen für die berechneten Subfunktionen:*

$$a = f_{x_1},\quad b = f_{\bar{x}_1},\quad c = f_{x_1x_2},\quad d = f_{x_1\bar{x}_2},\quad e = f_{\bar{x}_1x_2},\quad g = f_{\bar{x}_1\bar{x}_2},\quad h = f_{x_1x_2\bar{x}_3}$$

*Damit ergibt sich die folgende Darstellung von $BDD(f; x_1, x_2, x_3, x_4)$:*




*Manchmal ist es übersichtlicher, eine „abgerollte“ Version des Graphen zu benutzen, bei der Knoten so wiederholt werden, dass der Graph die spezielle Form eines Baumes annimmt. Für das Beispiel ergibt sich:*




**Beispiel 2.9** *Wir wählen die gleiche Funktion wie im vorangegangenen Beispiel, jedoch eine andere Reihenfolge der Variablen: $x_3, x_2, x_4, x_1$. Dann ergeben sich folgende Subfunktionen:*

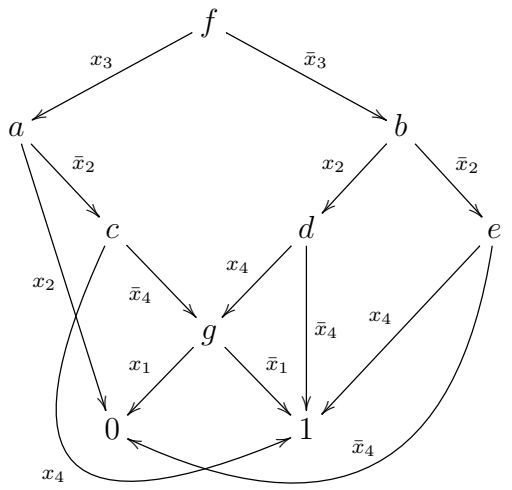
$$\begin{aligned}
f_{x_3} &= \langle \bar{x}_2x_4 + \bar{x}_1\bar{x}_2 \rangle \\
f_{x_3x_2} &= 0 \\
f_{\bar{x}_3x_2} &= \langle \bar{x}_4 + \bar{x}_1 \rangle \\
f_{x_3\bar{x}_2x_4} &= 1 \\
f_{\bar{x}_3x_2x_4} &= \langle \bar{x}_1 \rangle \\
f_{\bar{x}_3\bar{x}_2x_4} &= 1 \\
f_{x_3\bar{x}_2\bar{x}_4x_1} &= 0
\end{aligned}$$

$$\begin{aligned}
f_{\bar{x}_3} &= \langle \bar{x}_2x_4 + x_2\bar{x}_4 + \bar{x}_1x_2 \rangle \\
f_{x_3\bar{x}_2} &= \langle x_4 + \bar{x}_1 \rangle \\
f_{\bar{x}_3\bar{x}_2} &= \langle x_4 \rangle \\
f_{x_3\bar{x}_2\bar{x}_4} &= \langle \bar{x}_1 \rangle \\
f_{\bar{x}_3x_2\bar{x}_4} &= 1 \\
f_{\bar{x}_3\bar{x}_2\bar{x}_4} &= 0 \\
f_{x_3\bar{x}_2\bar{x}_4\bar{x}_1} &= 1
\end{aligned}$$

*Mit den abkürzenden Bezeichnungen*

$$a = f_{x_3}, \quad b = f_{\bar{x}_3}, \quad c = f_{x_3\bar{x}_2}, \quad d = f_{\bar{x}_3 x_2}, \quad e = f_{\bar{x}_3\bar{x}_2}, \quad g = f_{x_3\bar{x}_2\bar{x}_4}$$

*erhält man dann den folgenden Graph:*




Offensichtlich hat also die Reihenfolge der Variablen einen grossen Einfluss auf das enstehende BDD. Es sind derzeit keine brauchbaren Algorithmen bekannt, mit denen eine günstige Reihenfolge der Variablen bestimmen kann, bevor im Wesentlichen das vollständige BDD konstruiert ist.

# 3. Optimierung kombinatorischer Schaltungen

Basierend auf den in Kapitel 2 eingeführten theoretischen Grundlagen wenden wir uns in diesem Kapitel der Optimierung von kombinatorischen Schaltungen in Form von Schaltfunktionen zu. Zu diesem Thema werden zunächst wieder einige grundlegende Begriffe eingeführt, und im Anschluss daran werden dann verschiedene Verfahren vorgestellt.

## 3.1. Primimplikanten

In diesem Abschnitt wird zunächst der Begriff des „Primimplikanten“ eingeführt, als Vorbereitung für eine Reihe von Optimierungsverfahren für kombinatorische Schaltungen. Insbesondere wird vorgestellt, wie sämtliche Primimplikanten einer Schaltfunktion erzeugt werden können. Dieser Abschnitt ist bewusst ausführlich gehalten, da die hier vorgestellten Begriffe grundlegend für den algorithmischen Umgang mit Schaltfunktionen und Formeln sind.

Ausgangspunkt für alle Betrachtungen in diesem Abschnitt ist die folgende sehr einfache Beobachtung:

**Lemma 3.1** *Sei $n \in \mathbb{N}$ und sei $f = \langle Q_1 + \ldots + Q_m \rangle \in F_n$ eine Schaltfunktion, wobei $Q_1, \ldots, Q_n$ Produktterme sind. Dann gilt für alle $b \in \mathbb{B}^n$ und für alle $i = 1, \ldots, m$:*

$$\langle Q_i \rangle (b) = 1 \quad \Rightarrow \quad f(b) = 1$$

Mit anderen Worten besagt das Lemma: Für $f = \langle Q_1 + \ldots + Q_m \rangle$ gilt $\langle Q_i \rangle \to f$ für alle $i = 1, \ldots, m$. Wie man sich leicht überlegen kann, müssen unter den Produktttermen mit $\langle Q_i \rangle \to f$ minimale (d.h. „kürzeste“) existieren. Dies gibt Anlass zu folgender Definition.

**Definition 3.1 (Implikant, Primimplikant)** *Sei $n \in \mathbb{N}$, $\mathcal{V} = \{x_1, \ldots, x_n\}$, $t \in F_{\mathcal{V}}$ ein Produktterm und $f \in F_n$ eine Schaltfunktion.*

1. *$t$ heißt* Implikant *von $f$ wenn $\langle t \rangle \to f$.*
2. *$t$ heißt* Primimplikant *von $f$, wenn*
   a) *$t$ ein Implikant von $f$ ist, und*

*b) für alle Produktterme $s \in F_{\mathcal{V}}$*

$$\langle s \rangle \to f \quad \wedge \quad \langle t \rangle \to \langle s \rangle \quad \Rightarrow \quad s = t$$

*gilt.*

Zum Verständnis von $s = t$, sei an dieser Stelle daran erinnert, dass gemäß Definition 2.10 jede Variable in einem Produktterm höchstens einmal vorkommen darf, und dass nach Vereinbarung die Reihenfolge der Variablen keine Rolle spielt. Demzufolge ist also $\bar{x}_1 x_2 \bar{x}_1$ überhaupt kein Produktterm und $\bar{x}_1 x_2$ wird mit $x_2 \bar{x}_1$ identifiziert.

Die Bedeutung von $\langle t \rangle \to \langle s \rangle$ in der obigen Definition ist einfacher als es zunächst den Anschein haben mag. Wie im folgenden Lemma dargestellt, ist in diesem Fall der Term $s$ eine „Verkürzung“ des Terms $t$ (d.h. $s$ ist syntaktischer Bestandteil von $t$).

**Lemma 3.2** *Es seien $t = x_{i_1}^{e_1} \dots x_{i_m}^{e_m}$ und $s = x_{j_1}^{r_1} \dots x_{j_n}^{r_n}$ Produktterme über einer beliebigen Variablenmenge. Wenn $\langle t \rangle \to \langle s \rangle$, dann gilt:*

1. *$\{j_1, \dots, j_n\} \subseteq \{i_1, \dots, i_m\}$.*
2. *Wenn $i_k = j_k$, so auch $e_k = r_k$.*

Wie zuvor erwähnt, können in Produkttermen Variablenwiederholungen nicht vorkommen. Das bedeutet insbesondere, dass $P \cdot Q$ nicht zwangsläufig ein Produktterm ist, selbst wenn $P$ und $Q$ solche sind. Wegen der Idempotenz, die in Booleschen Algebren gilt, liegt es aber nahe, Wiederholungen einfach zu streichen und auf diese Weise aus $P \cdot Q$ einen Produktterm zu machen. Beispielsweise indem $x_1 x_2 \bar{x}_3 \cdot x_2 x_4$ zu $x_1 x_2 \bar{x}_3 x_4$ umgeformt wird. Siehe dazu die folgende Definition.

**Definition 3.2** *Es seien $P = x_{i_1}^{e_1} \dots x_{i_m}^{e_m}$ und $Q = x_{j_1}^{r_1} \dots x_{j_n}^{r_n}$ Produktterm.*

$$P * Q \quad := \quad \begin{cases} 0, & \text{falls } i_k = j_l \text{ und } e_k \neq r_l \\ & \text{für } k \in \{1, \dots, m\}, l \in \{1, \dots, n\} \\ x_{d_1}^{a_1} \dots x_{d_t}^{a_t}, & \text{mit } \{d_1, \dots, d_t\} = \{i_1, \dots, i_m\} \cup \{j_1, \dots, j_n\} \\ & \text{und } a_k = \begin{cases} e_l, & \text{falls } d_k = i_l \\ r_l, & \text{falls } d_k = j_m \end{cases} \quad \text{sonst} \end{cases}$$

Mit dieser vereinfachten Schreibweise läßt sich nun einer der zentralen Sätze für die Erzeugung von Primimplikanten sehr kompakt formulieren.

**Satz 3.1** *Seien $f, f_1, \dots, f_k$ Schaltfunktionen mit $f = \prod_{i=1}^{k} f_i$ und sei $P$ ein Primimplikant von $f$. Dann existieren Primimplikanten $P_i$ von $f_i$ für $i = 1, \dots, k$, so dass $P = P_1 * \dots * P_k$ gilt.*

**Beweis:** Aus $\langle P \rangle (b) = 1$ folgt $f(b) = 1$ und daraus folgt $f_i(b) = 1$ für alle $i = 1, \ldots, k$ und alle $b \in \mathbb{B}^n$. Daher ist $P$ ein Implikant jeder Funktion $f_i$. Also muss für jedes $i$ mindestens eine Verkürzung $P_i$ existieren, so dass $P_i$ Primimplikant von $f_i$ ist. Aus $\langle P \rangle \rightarrow \langle P_i \rangle$ folgt aber $\langle P \rangle \rightarrow \langle P_1 * \ldots * P_k \rangle$. Andererseits folgt aus $\langle P_1 * \ldots * P_k \rangle (b) = 1$ auch $\langle P_i \rangle (b) = 1$ und somit $f_i(b) = 1$ für alle $i = 1, \ldots, k$. Wegen $f = \prod_{i=1}^{k} f_i$ können wir auf $f(b) = 1$ schliessen und haben damit $\langle P_1 * \ldots * P_k \rangle \rightarrow f$ gezeigt. Weil $P$ aber nach Voraussetzung ein Primimplikant von $f$ ist, folgt schliesslich $P = P_1 * \ldots * P_k$. □

Aus dem gerade bewiesenen Satz lässt sich dann eine einfache Methode zur Erzeugung von Primimplikanten ableiten. Wenn $f = \prod_{i=1}^{k} f_i$ ist und alle Primimplikanten der $f_i$ bekannt sind, dann bildet man alle Produktterme der Form $P_1 * \ldots * P_k$. Nicht alle so erzeugten Produktterme sind Primimplikanten, weil aber auch alle Primimplikanten erzeugt wurden, sind die übrigen Terme leicht zu erkennen, da sie ja Verlängerungen von Primimplikanten sind.

Der nächste Abschnitt beschäftigt sich detailiert mit der Erzeugung von Primimplikanten für Schaltfunktionen unter Ausnutzung unterschiedlicher Techniken.

## 3.2. Erzeugung von Primimplikanten

In diesem Abschnitt werden verschiedene Methoden zur Erzeugung von Primimplikanten vorgestellt, welche sich mehr oder weniger direkt aus den im vorangegangenen Abschnitt dargestellten Grundlagen herleiten.

### 3.2.1. Ausmultiplizieren der Maxtermnormalform

Wenn eine Schaltfunktion $f$ durch ihre Wertetabelle gegeben ist, dann lässt sich sehr leicht eine Darstellung $f = \prod_{i=1}^{k} f_i = \prod_{i=1}^{k} \langle F_i \rangle$ ablesen, wobei $k$ die Anzahl der Nullstellen von $f$ bezeichnet. Jedes $F_i$ muss dann von der Form $F_i = x_{i_1}^{e_1} + \ldots + x_{i_n}^{e_n}$ sein. Nun ist $F_i$ Implikant von $f_i$ und nicht weiter verkürzbar. Damit ist $F_i$ also ein Primimplikant von $f_i$. Das Bilden aller Primimplikanten der Form $P_1 * \ldots * P_k$ bedeutet aber nichts anderes als „Ausmultiplizieren“ der Maxtermnormalform.

**Beispiel 3.1** *Zur Verdeutlichung greifen wir erneut die bereits zuvor betrachtete Funktion $f \in F_4$ auf, die durch folgende Wertetabelle definiert ist.*

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $f$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

*Als Maxtermnormalform lesen wir*

$$\begin{aligned} f &= \left\langle \prod_{i=1}^{8} F_i \right\rangle \\ &= \left\langle \begin{array}{l} (x_1+x_2+x_3+x_4)\ (x_1+\bar{x}_2+\bar{x}_3+x_4) \\ (x_1+\bar{x}_2+\bar{x}_3+\bar{x}_4)\ (\bar{x}_1+x_2+x_3+x_4) \\ (\bar{x}_1+x_2+\bar{x}_3+x_4)\ (\bar{x}_1+\bar{x}_2+x_3+\bar{x}_4) \\ (\bar{x}_1+\bar{x}_2+\bar{x}_3+x_4)\ (\bar{x}_1+\bar{x}_2+\bar{x}_3+\bar{x}_4) \end{array} \right\rangle \end{aligned}$$

*ab. „Ausmultiplizieren“ ergibt folgendes:*

$$\begin{aligned} & (x_1+x_2+x_3+x_4)(x_1+\bar{x}_2+\bar{x}_3+x_4) \\ \equiv\ & x_1+x_1\bar{x}_2+x_1\bar{x}_3+x_1x_4+x_1x_2+x_2\bar{x}_3+x_2x_4 \\ & +x_1x_3+\bar{x}_2x_3+x_3x_4+x_1x_4+\bar{x}_2x_4+\bar{x}_3x_4+x_4 \end{aligned}$$

*Nun ist z.B. $x_1\bar{x}_2$ eine Verlängerung von $x_1$, also kein Primimplikant von $F_1 \cdot F_2$. Derartige Verlängerungen können sofort unterdrückt werden. Wir erhalten:*

$$\begin{aligned} G_1 &= F_1 \cdot F_2 \equiv x_1+x_4+x_2\bar{x}_3+\bar{x}_2x_3 \\ G_2 &= F_3 \cdot F_4 \equiv x_1x_2+x_1x_3+x_1x_4+\bar{x}_1\bar{x}_2+\bar{x}_2x_3+\bar{x}_2x_4 \\ & \qquad +\bar{x}_1\bar{x}_3+x_2\bar{x}_3+\bar{x}_3x_4+\bar{x}_1\bar{x}_4+x_2\bar{x}_4+x_3\bar{x}_4 \\ G_3 &= F_5 \cdot F_6 \equiv \bar{x}_1+x_2x_3+x_2\bar{x}_4+\bar{x}_2\bar{x}_3+\bar{x}_3\bar{x}_4+\bar{x}_2x_4+x_3x_4 \\ G_4 &= F_7 \cdot F_8 \equiv \bar{x}_1+\bar{x}_2+\bar{x}_3 \\ H_1 &= G_1 \cdot G_2 \equiv x_1x_2+x_1x_3+x_1x_4+\bar{x}_2x_4+\bar{x}_3x_4+x_2\bar{x}_3+\bar{x}_2x_3 \\ H_2 &= G_3 \cdot G_4 \equiv \bar{x}_1+\bar{x}_2\bar{x}_3+\bar{x}_2x_4+\bar{x}_3\bar{x}_4 \\ & H_1 \cdot H_2 \equiv \bar{x}_2x_4+\bar{x}_1\bar{x}_3x_4+\bar{x}_1x_2\bar{x}_3+x_2\bar{x}_3\bar{x}_4+\bar{x}_1\bar{x}_2x_3 \end{aligned}$$

*In dieser Summe stehen nun sämtliche Primimplikanten von $f$. Wie sich der Leser leicht durch Bildung anderer Teilprodukte überzeugen kann, ist der Rechenaufwand in hohem Maße abhängig von den gewählten Teilprodukten.*

Diese Methode zur Primimplikantenerzeugung erscheint einfach und universell. Aber wie bereits anhand des vorangegangenen Beispiels ersichtlich werden dürfte ist der Rechenaufwand bereits für Funktionen $f \in F_n$ mit kleinem $n$ schon erheblich. Darüberhinaus setzt dieses Verfahren voraus, dass die Maxtermnormalform bekannt ist, was exponentiellen Speicherbedarf impliziert.

### 3.2.2. Die Methode von Quine und McCluskey

Man sollte sich jedoch nicht täuschen lassen, die Manipulation von Formeln, wie sie bisher vorgestellt wurde um Primpimplikanten zu erzeugen, ist in der Regel sehr aufwendig, selbst wenn die Formeln geschickt kodiert werden. Wir wollen daher als nächstes das klassische Verfahren einführen, das sogenannte *Verfahren nach Quine und McClusky (QMCV)*, benannt nach Willard Van Orman Quine und Edward J. McCluskey, denen das Verfahren zugeschrieben wird. Der Ausgangspunkt ist die Mintermnormalform und die benötigte Operation ist der *Simple Consensus*, wie in Lemma 2.6 dargestellt.

**Definition 3.3 (Simpler Consensus)** *Es seien $Q$ und $R$ Produktterme über einer beliebigen Variablenmenge $\mathcal{V}$.*

$$d(Q,R) \;:=\; \begin{cases} S, & \text{falls } Q = aS \text{ und } R = \bar{a}S \text{ für ein beliebiges } a \in \mathcal{V} \\ Q, & \text{sonst} \end{cases}$$

Mit dieser Definition lässt sich dann das Verfahren zur Primimplikantenerzeugung selbst leicht beschreiben.

**Algorithmus 3.1 (Methode von Quine und McCluskey)** *Es seien $Q_1, \ldots, Q_n$ Minterme und $f = \left\langle \sum_{i=1}^{n} Q_i \right\rangle$.*

1. $L_0 := \{Q_1, \ldots, Q_n\}$
2. $i := 0$
3. $L_{i+1} := L_i \cup \{d(R_j, R_k) \mid R_j, R_k \in L_i\}$
4. *Wiederhole* 3 *mit* $i := i + 1$ *falls* $L_{i+1} \neq L_i$, *sonst weiter mit* 5.
5. *Streiche alle Produktterme, die Verlängerungen anderer Produktterme in $L_{i+1}$ sind. Bezeichne das Ergebnis mit $L_E$.*

Der Leser sollte bereits vertraut sein mit diesem Verfahren, und es sollte dementsprechend intuitiv klar sein, dass diese Methode genau die Primimplikanten einer Funktion erzeugt. Der folgende Satz formalisiert diese Intuition.

**Satz 3.2** *Wird Algorithmus 3.1 auf eine Liste $L_0 = \{Q_1, \ldots, Q_n\}$ angewandt, deren Elemente Minterme sind, dann enthält $L_E$ genau die Primimplikanten von $\langle Q_1 + \ldots + Q_n \rangle$.*

**Beweis:** Es genügt zu zeigen, dass $L_i$ alle Implikanten von $\{Q_1, \ldots, Q_n\}$ enthält, deren Länge mindestens $m - i$ ist, wobei $m$ die Anzahl der verwendeten Schaltvariablen ist. Dies lässt sich leicht durch Induktion über $i$ zeigen.

1. Für $i = 0$ gilt $L_i = L_0$, und $L_0$ enthält nach Voraussetzung die Minterme.

2. Sei $R$ ein Implikant der Länge $m-(i+1)$ und sei $x$ nicht in $R$ enthalten. Daraus folgt, dass sowohl $xR$ als auch $\bar{x}R$ Implikanten der Länge $m-i$ sind. Nach Induktionsvoraussetzung sind $xR$ und $\bar{x}R$ damit in $L_i$. Also ist $d(xR, \bar{x}R) = R \in L_{i+1}$.□

**Beispiel 3.2** *Betrachten wir beispielhaft die Funktion*

$$f = \left\langle \begin{array}{l} \bar{w}\bar{x}\bar{y}\bar{z} + \bar{w}x\bar{y}z + \bar{w}xy\bar{z} + \bar{w}xyz + w\bar{x}\bar{y}z \\ +w\bar{x}y\bar{z} + wx\bar{y}z + wxy\bar{z} + wxyz \end{array} \right\rangle.$$

*Mit dem Verfahren von Quine und McCluskey berechen wir zunächst alle Implikanten von $f$.*

1. $L_0 = \{\bar{w}\bar{x}\bar{y}\bar{z}, \bar{w}x\bar{y}z, \bar{w}xy\bar{z}, \bar{w}xyz, w\bar{x}\bar{y}z, w\bar{x}y\bar{z}, wx\bar{y}z, wxy\bar{z}, wxyz\}$
2. $L_1 = L_0 \cup \{\bar{w}xz, x\bar{y}z, \bar{w}xy, xy\bar{z}, \bar{w}\bar{y}z, wy\bar{z}, xyz, wxz, wxy\}$
3. $L_2 = L_1 \cup \{xz, xy\}$
4. $L_3 = L_2$

   *Nun sind alle Implikanten von $f$ erzeugt und es müssen Verlängerungen gestrichen werden. Als Ergebnis erhalten wir die Primimplikantenmenge*
5. $L_E = \{\bar{w}\bar{x}\bar{y}\bar{z}, w\bar{y}z, wy\bar{z}, xz, xy\}$.

Die Methode von Quine und McCluskey erscheint deshalb so einfach und effizient, weil die verwendeten Operationen sehr einfach sind. Allerdings ist es wohl offensichtlich sehr aufwendig, zunächst *alle* Implikanten einer Funktion zu erzeugen, um dann die Primimplikanten übrig zu behalten. Außerdem liegen Funktionen nicht unbedingt immer in Mintermnormalform oder in Form einer Wertetabelle vor. In solchen Fällen ist es überraschend aufwendig, die Ausgangsform für den Algorithmus, nämlich die Mintermnormalform, zu erzeugen.

Allgemein gilt, dass ein Algorithmus, welcher das Verfahren von Quine und McCluskey implementiert, immer mindestens $\mathcal{O}(3^n)$ Schritte benötigt, wobei $n$ die Anzahl der Variablen ist. Die genaue Komplexität hängt stark von der konkreten Implementation ab; hier kann mit Hilfe von geschickten Optimierungen eine Komplexität nahe $\mathcal{O}(3^n)$ erreicht werden. In jedem Fall ist damit jedoch gezeigt, dass das Verfahren für eine praktische Anwendung mit einem realistischen $n$ nicht in Frage kommt.

### 3.2.3. Primimplikantenerzeugung mit BDDs

Basierend auf der Erkenntnis, dass das Verfahren von Quine und McCluskey zwar auf den ersten Blick einfach genug aussieht, für die Praxis jedoch untauglich ist, da es ausgehend von einer Mintermnormalform arbeitet und zunächst alle Implikanten einer Funktion erzeugt, was gerade bei Schaltungen mit vielen Eingängen zu einer unrealistischen Laufzeit

führt, betrachten wir in diesem Abschnitt eine Modifikation des Verfahrens, die sogenannte *Baummethode* nach Reusch. Grundlage für diese Modifikation bildet der folgende Satz.

**Satz 3.3** *Sei $f$ eine Schaltfunktion, $x$ eine beliebige Variable und $P$ ein Primimplikant von $f$. Dann gilt eine der folgenden Aussagen*

1. $P = xP_x$
2. $P = \bar{x}P_{\bar{x}}$
3. $P = P_{\bar{x}} * P_x$,

*wobei jeweils $P_x$ ein Primimplikant von $f_x$ und $P_{\bar{x}}$ ein Primimplikant von $f_{\bar{x}}$ ist.*

Diesen Satz kann man nun direkt für eine Methode zur Erzeugung sämtlicher Primimplikanten einer Schaltfunktion benutzen.

**Algorithmus 3.2 (Baummethode)** *Sei $f = \langle F \rangle$ eine beliebige Schaltfunktion, wobei $F$ eine beliebige Schaltformel ist, die nicht notwendigerweise in Normalform vorliegen muss.*

1. *Wähle eine Variable $x$, welche in $F$ vorkommt.*
2. *Erzeuge rekursiv alle Primimplikanten von $\langle F \rangle_x$ und $\langle F \rangle_{\bar{x}}$.*
3. *Seien $P_1, \ldots, P_n$ die Primimplikanten von $\langle F \rangle_{\bar{x}}$ und $Q_1, \ldots, Q_m$ die Primimplikanten von $\langle F \rangle_x$.*

   *Bilde $\bar{x}P_1, \ldots, \bar{x}P_n, xQ_1, \ldots, xQ_m$ und $P_i * Q_j$ für $i = 1, \ldots, n$, $j = 1, \ldots, m$.*
4. *Streiche Verlängerungen.*

Es handelt sich hierbei um das Schema eines rekursiven Algorithmus, der stets terminiert, da bei der Suche nach Primimplikanten von Subfunktionen die Variablen einmal „verbraucht“ sein müssen. Weiterhin ist zu bemerken, dass wir schon $\langle F \rangle_x = \langle F_x \rangle$ gezeigt haben und daher mit Formeln arbeiten können.

Dabei macht es aber einen Unterschied, welche Umformungen an $F_x$ wir erlauben. Wir werden dies an einem Beispiel demonstrieren. Letztlich müssen wir natürlich noch festlegen, wann die Primimplikanten einer Funktion bekannt sind. Die einfachste Menge von Regeln ist die Folgende.

1. Stop-Regeln: Wenn $f_t = 0$ oder $f_t = 1$, wird nicht weiterentwickelt.
2. Vereinfachungs-Regeln:
   a) $1 + P \equiv 1$

b) $0 + P \equiv P$

Das folgende Beispiel wird zunächst ausschliesslich mit diesen Regeln entwickelt.

**Beispiel 3.3** *Wir wählen das bereits zuvor für die BDD-Konstruktion verwendete Beispiel der Funktion*

$$f \;=\; \langle \bar{x}_2 x_4 + x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 \rangle \;=\; \langle F \rangle \,.$$

*Entwickeln wir diese zunächst nach $x_1$, also die Zerlegung $f = x_1 \langle F_1 \rangle + \bar{x}_1 \langle F_0 \rangle$, so erhalten wir*

$$F_1 \;=\; \bar{x}_2 x_4 + x_2 \bar{x}_3 \bar{x}_4 \qquad \textit{und} \qquad F_0 \;=\; \bar{x}_2 x_4 + x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_2 x_3 + x_2 \bar{x}_3.$$

*Entwickeln wir dieses Ergebnis weiter nach $x_2$, so erhalten wir $\langle F_1 \rangle = x_2 \langle F_{11} \rangle + \bar{x}_2 \langle F_{10} \rangle$ und $\langle F_0 \rangle = x_2 \langle F_{01} \rangle + \bar{x}_2 \langle F_{00} \rangle$ mit*

$$\begin{array}{lcl@{\qquad}l@{\qquad}lcl} F_{11} &=& \bar{x}_3 \bar{x}_4, & & F_{10} &=& x_4, \\ F_{01} &=& \bar{x}_3 \bar{x}_4 + \bar{x}_3 & \textit{und} & F_{00} &=& x_4 + x_3. \end{array}$$

*Nach einer weiteren Entwicklung dieses Ergebnisses nach $x_3$ erhalten wir schliesslich*

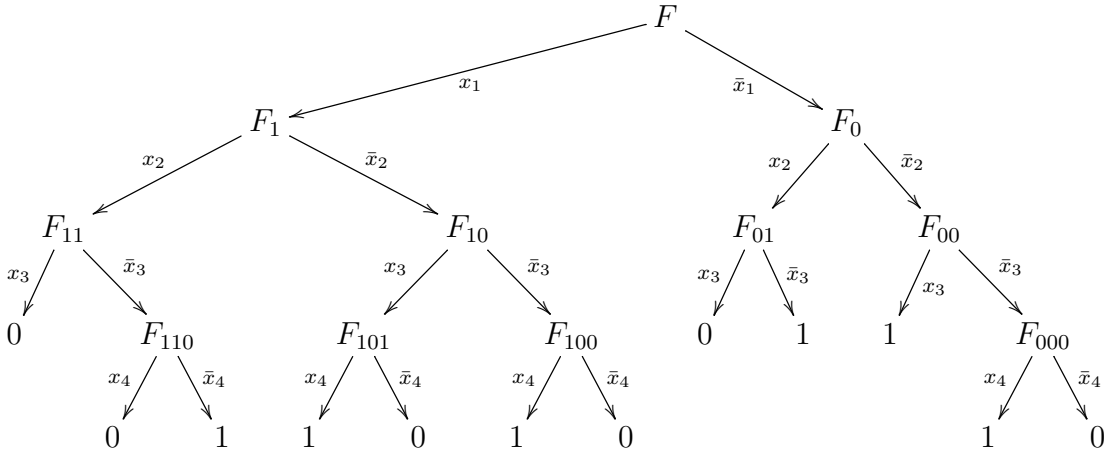
$$\begin{array}{lcl@{\qquad}l@{\qquad}lcl} F_{111} &=& 0, & & F_{110} &=& \bar{x}_4, \\ F_{101} &=& x_4, & & F_{100} &=& x_4, \\ F_{011} &=& 0, & & F_{010} &=& \bar{x}_4 + 1 \equiv 1, \\ F_{001} &=& x_1 + 1 \equiv 1 & \textit{und} & F_{000} &=& x_4, \end{array}$$

*wo dann nur noch die von 0 und 1 verschiedenen Subfunktionen nach $x_4$ weiterentwickelt werden müssen, so dass sich schlussendlich*

$$\begin{array}{lcl@{\qquad}l@{\qquad}lcl} F_{1101} &=& 0, & & F_{1100} &=& 1, \\ F_{1011} &=& 1, & & F_{1010} &=& 0, \\ F_{1001} &=& 1, & & F_{1000} &=& 0, \\ F_{0001} &=& 1 & \textit{und} & F_{0000} &=& 0 \end{array}$$

*ergeben.*

*Der Zusammenhang zwischen den erzeugten Subfunktionen lässt sich dann in folgendem Baum darstellen, was auch den Namen der Methode erklärt.*



*Nun lassen sich die Primimplikanten für die Subfunktionen „bottom-up“ berechnen. Also z.B. für $F_{110}$ auf folgende Weise: $0 \cdot x_4$, $1 \cdot \bar{x}_4$ und $0 \cdot 1$. Im Ergebnis dementsprechend, wenn wir die Primimplikanten aufsummieren, $F_{110} \equiv \bar{x}_4$. Ebenso ergeben sich*

$$F_{101} \equiv x_4, \qquad F_{100} \equiv x_4 \quad \text{und} \quad F_{000} \equiv x_4.$$

*Für die nächste Ebene greifen wir $F_{10}$ heraus. Gemäss der Methode ergibt sich zunächst $F_{10} \equiv x_3x_4 + \bar{x}_3x_4 + x_4$ und schliesslich $F_{10} \equiv x_4$. Für die übrigen erhalten wir*

$$F_{11} \equiv \bar{x}_3\bar{x}_4, \qquad F_{01} \equiv \bar{x}_3 \quad \text{und} \quad F_{00} \equiv x_3 + \bar{x}_3x_4 + x_4 \equiv x_3 + x_4.$$

*Der nächste Schritt bringt uns zu*

$$\begin{aligned} F_1 &\equiv x_2\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 \\ F_0 &\equiv x_2\bar{x}_3 + \bar{x}_2x_3 + \bar{x}_2x_4 + \bar{x}_3x_4 \end{aligned}$$

*und daraus schliesslich erhalten wir die Primimplikanten von $f$ durch*

$$\begin{aligned} F &\equiv x_1x_2\bar{x}_3x_4 + x_1\bar{x}_2x_4 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_2x_4 \\ &\quad + x_1\bar{x}_3x_4 + x_2\bar{x}_3\bar{x}_4 + \bar{x}_2x_3x_4 + \bar{x}_2x_4 + \bar{x}_2\bar{x}_3x_4 \\ &\equiv \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1\bar{x}_3x_4 + x_2\bar{x}_3x_4 + x_2\bar{x}_3\bar{x}_4 + \bar{x}_2x_4. \end{aligned}$$

## 3.3. Minimale Summen

Im vorhergigen Abschnitt wurden Methoden vorgestellt, mit denen sich alle Primimplikanten einer Schaltfunktion erzeugen lassen. In diesem Abschnitt nun wird eine Optimierungsaufgabe für Schaltfunktionen bzw. deren Realisierung durch Gatterschaltungen untersucht. Hierzu wird angenommen, dass `UND`-Gatter und `ODER`-Gatter mit einer beliebigen Anzahl von Eingängen zur Verfügung stehen und die Eingänge der `UND`-Gatter auch komplementiert zur Verfügung stehen.

Die Optimierungsaufgabe besteht nun darin für eine gegebene Schaltfunktion eine möglichst günstige Realisierung als Gatterschaltung zu finden. Das bedeutet aber nichts anderes, als dass zu einer gegebenen Schaltfunktion eine Summe von Produkten zu finden ist, die möglichst „billig“ zu realisieren ist, wobei ein Kostenkriterium noch zu definieren ist. Für die hier betrachtete Aufgabenstellung genügt es, die zugelassenen Kostenkriterien durch die folgenden zwei Eigenschaften zu charakterisieren.

**Definition 3.4 (Kostenkriterium)** *Sei $F$ eine Schaltformel, $k(F) \in \mathbb{R}$ sind die Kosten von $F$. $k$ ist* zugelassenes Kostenkriterium*, wenn gilt:*

1. *Für Summen $Q_1 + \ldots + Q_m$ und $P_1 + \ldots + P_n$ mit $\{P_1, \ldots, P_n\} \subseteq \{Q_1, \ldots, Q_m\}$ gilt $k(Q_1 + \ldots + Q_m) > k(P_1 + \ldots + P_n)$.*
2. *Für Produkte $P = x_1^{e_1} \ldots x_m^{e_m}$ und $Q = y_1^{g_1} \ldots y_n^{g_n}$ mit $m < n$ gilt $k(P) < k(Q)$.*

In Worten ausgedrückt ist die Bedeutung dieser Eigenschaft einfach:

1. Kürzere Summen sind billiger als längere.

2. Kürzere Produkte sind billiger als längere.

**Definition 3.5 (Irredundante Summe)** $Q_1 + \ldots + Q_n$ *heisst* irredundante Summe*, wenn* $\langle Q_1 + \ldots + Q_n \rangle \neq \langle Q_{i_1} + \ldots + Q_{i_m} \rangle$ *für alle* $\{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$.

Damit lässt sich nun der zentrale Satz dieses Abschnitts formulieren und beweisen.

**Satz 3.4** *Sei $f$ eine Schaltfunktion. Wenn $S = P_1 + \ldots + P_n$ eine kostenminimale Realisierung von $f$ ist, wobei $P_1, \ldots, P_n$ Produktterme sind, dann ist $S$ eine irredundante Summe von Primimplikanten von $f$ und es gilt $f = \langle P_1 + \ldots + P_n \rangle$.*

Dieser Zusammenhang ist elementar für das Verständnis der in diesem Abschnitt behandelten Optimierungsaufgaben. Eine kostenminimale, zweistufige Realisierung ist immer eine irredundante Summe von Primimplikanten. Damit ist nun auch klar, warum das Erzeugen von Primimplikanten von Schaltfunktionen für die Schaltungssynthese so wichtig ist.

Im Umkehrschluss bedeutet dieser Zusammenhang, dass es zur kostenminimalen, zweistufigen Schaltungssynthese ausreicht, alle irredundanten Summen einer Schaltfunktion zu bilden, und dann aus dieser Menge von Summen die kostengünstigsten zu bestimmen. Diese bezeichnet man dann als die *minimalen Summen.*

**Algorithmus 3.3 (Konstruktion minimaler Summen)**

1. *Erzeugung aller Primimplikanten.*

2. *Konstruktion aller irredundanten Summen.*

3. *Auswahl der minimalen Summen.*

Der erste Schritt des Algorithmus wurde im vorangegangenen Abschnitt ausführlich besprochen. Der zweite Schritt ist Gegenstand dieses Abschnitts. Schritt 3 wird durch reine Inspektion aller irredundanten Summen durchgeführt.

Zunächst beschäftigen wir uns mit dem Spezialfall, dass die Mintermnormalform bekannt ist. Der nachfolgende Satz enthält in diesen Fall die Lösung für das Problem.

**Satz 3.5** *Seien $I, J \subseteq \mathbb{N}$. Seien weiter $\{m_i \mid i \in I\}$ Minterme und $\{P_j \mid j \in J\}$ Produktterme. Dann gilt* $\left\langle \sum_{i \in I} m_i \right\rangle = \left\langle \sum_{j \in J} P_j \right\rangle$ *genau dann, wenn*

1. *für jedes $i \in I$ ein $j \in J$ existiert mit $\langle m_i \rangle \rightarrow \langle P_j \rangle$, und*

*2. für jedes $j \in J$ gilt $\langle P_j \rangle \rightarrow \left\langle \sum_{i \in I} m_i \right\rangle$.*

Wenn $f = \left\langle \sum_{i \in I} m_i \right\rangle$ und $P_1, \ldots, P_n$ alle Primimplikanten von $f$ sind, dann sind die beiden Bedingungen *1.* und *2.* des vorangegangenen Satzes aus offensichtlichen Gründen erfüllt. Für irredundante Summen von Primimplikanten gilt es nun Teilmengen $\{j_1, \ldots, j_k\} \subseteq \{1, \ldots, n\}$ zu finden, die die Bedingungen erfüllen, die aber keine echten Teilmengen mehr besitzen, die noch die Bedingungen erfüllen.

Interessant ist dabei insbesondere die erste Bedingung, da die zweite Bedingung selbstverständlich für jedes $P_j$ erfüllt ist. Damit lautet die Aufgabenstellung zum Auffinden von irredundanten Summen wie folgt:

> Finde alle nicht weiter verkleinerbaren Teilmengen $\{j_1, \ldots, j_k\}$ von $\{1, \ldots, n\}$, so dass für jedes $i \in I$ ein $j_l$ existiert mit $\langle m_i \rangle \rightarrow \langle P_{j_l} \rangle$.

Hierbei handelt es sich um einen Spezialfall des sogenannten „Überdeckungsproblems“, welches für endliche Mengen wie folgt allgemein definiert ist.

**Definition 3.6 (Überdeckungsproblem)** *Gegeben sei eine endliche Menge $M$ und ein endliches System von Teilmengen $S_i \subseteq M$, $i \in I$. Gesucht sind Teilmengen $U \subseteq M$ mit den folgenden Eigenschaften:*

1. *$U \cap S_i \neq \emptyset$ für alle $i \in I$.*
2. *Wenn $V \subset U$, dann existiert ein $i \in I$ mit $V \cap S_i = \emptyset$.*

**Beispiel 3.4** *Die Schaltfunktion $f$ sei als Summe der Minterme gegeben, die als Zeilenbezeichnung in der nachfolgend dargestellten Matrix zu finden sind. Als Spaltenbezeichnungen werden die Primimplikanten für $f$ benutzt. Jedem Minterm $m_i$ wird nun die Menge von Primimplikanten $P_j$ zugeordnet, für die $\langle m_i \rangle \rightarrow \langle P_j \rangle$ gilt, d.h. für die $P_j$ eine Verkürzung von $m_i$ ist.*

| | $ab$ | $cd$ | $a\bar{c}$ | $d\bar{e}$ | $ad$ | $ae$ | $\bar{b}d$ | $\bar{b}e$ | $\bar{a}\bar{b}c$ | $\bar{c}\bar{d}e$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $abcde$ | × | × | | | × | × | | | | |
| $abcd\bar{e}$ | × | × | | × | × | | | | | |
| $abc\bar{d}e$ | × | | | | | × | | | | |
| $abc\bar{d}\bar{e}$ | ⊗ | | | | | | | | | |
| $ab\bar{c}de$ | × | | × | | × | × | | | | |
| $ab\bar{c}d\bar{c}$ | × | | × | × | × | | | | | |
| $ab\bar{c}\bar{d}e$ | × | | × | | | × | | | | × |
| $ab\bar{c}\bar{d}\bar{e}$ | × | | × | | | | | | | |
| $a\bar{b}cde$ | | × | | | × | × | × | × | | |
| $a\bar{b}cd\bar{e}$ | | × | | × | × | | × | | | |
| $a\bar{b}c\bar{d}e$ | | | | | | × | | × | | |
| $a\bar{b}c\bar{d}\bar{e}$ | | | | | | | | | | |
| $a\bar{b}\bar{c}de$ | | | × | | × | × | × | × | | |
| $a\bar{b}\bar{c}d\bar{e}$ | | | × | × | × | | × | | | |
| $a\bar{b}\bar{c}\bar{d}e$ | | | × | | | × | | × | | × |
| $a\bar{b}\bar{c}\bar{d}\bar{e}$ | | | ⊗ | | | | | | | |
| $\bar{a}bcde$ | | ⊗ | | | | | | | | |
| $\bar{a}bcd\bar{e}$ | | × | | × | | | | | | |
| $\bar{a}bc\bar{d}e$ | | | | | | | | | | |
| $\bar{a}bc\bar{d}\bar{e}$ | | | | | | | | | | |
| $\bar{a}b\bar{c}de$ | | | | | | | | | | |
| $\bar{a}b\bar{c}d\bar{e}$ | | | | ⊗ | | | | | | |
| $\bar{a}b\bar{c}\bar{d}e$ | | | | | | | | | | ⊗ |
| $\bar{a}b\bar{c}\bar{d}\bar{e}$ | | | | | | | | | | |
| $\bar{a}\bar{b}cde$ | | × | | | | | × | × | × | |
| $\bar{a}\bar{b}cd\bar{e}$ | | × | | × | | | × | | × | |
| $\bar{a}\bar{b}c\bar{d}e$ | | | | | | | | × | × | |
| $\bar{a}\bar{b}c\bar{d}\bar{e}$ | | | | | | | | | ⊗ | |
| $\bar{a}\bar{b}\bar{c}de$ | | | | | | | × | × | | |
| $\bar{a}\bar{b}\bar{c}d\bar{e}$ | | | | × | | | × | | | |
| $\bar{a}\bar{b}\bar{c}\bar{d}e$ | | | | | | | | × | | × |
| $\bar{a}\bar{b}\bar{c}\bar{d}\bar{e}$ | | | | | | | | | | |

*Zum Beispiel gilt* $a\bar{b}\bar{c}d\bar{e} \rightarrow \{a\bar{c}, d\bar{e}, ad, \bar{b}d\}$. *In der Matrix sind zeilenweise die zugeordneten Mengen durch* × *markiert. Einige Mengen bestehen nur aus einem einzelnen Element (in der entsprechenden Zeile gibt es nur eine Markierung* ⊗*). Diese Elemente* müssen *in jeder überdeckenden Menge U vorkommen. Sie heissen allgemein Kernelemente oder hier im Speziallfall* Kernimplikanten. *In der Matrix haben wir diese Element mit* ⊗ *gekennzeichnet. Wenn nun* $P_j$ *ein Kernimplikant ist und* $\langle m_i \rangle \rightarrow \langle P_j \rangle$ *gilt, dann muss für* $m_i$ *kein weiteres überdeckendes Element benutzt werden.*

*Wenn also in unserem Beispiel ab ein Kernimplikant ist, weil er die einzige Überde-*

*ckung von $abc\bar{d}\bar{e}$ darstellt, sind damit auch abcde bis $ab\bar{c}\bar{d}\bar{e}$ überdeckt. Diese Zeilen fallen also für eine weitere Betrachtung weg. Nachdem wir so alle Kernimplikanten ab, cd, $a\bar{c}$, $d\bar{e}$, $\bar{a}\bar{b}c$ und $\bar{c}\bar{d}e$ durchgemustert haben, bleibt nur noch folgende Matrix übrig:*

| | $ad$ | $ae$ | $\bar{b}d$ | $\bar{b}e$ |
|---|---|---|---|---|
| $a\bar{b}cd\bar{e}$ | | × | | × |
| $\bar{a}\bar{b}\bar{c}de$ | | | × | × |

*Interessanterweise trägt ad nicht mehr zu einer irrendundanten Überdeckung bei: Jeder Minterm, der von ad überdeckt werden könnte, ist bereits von einem oder mehreren Kernimplikanten überdeckt worden. Solche Primimplikanten heissen* absolut eliminierbar*. Als irredundante Summen erhalten wir:*

1. $ab + cd + a\bar{c} + d\bar{e} + \bar{a}\bar{b}c + \bar{c}\bar{d}e + \bar{b}e$
2. $ab + cd + a\bar{c} + d\bar{e} + \bar{a}\bar{b}c + \bar{c}\bar{d}e + ae + \bar{b}d$

*Offensichtlich ist die erste Realisierung die kostengünstigere.*

Die Begriffe *Kernimplikant* und *absolut eliminierbar*, die bisher nur informal definiert und benutzt wurden, werden nun im Folgenden in einer Definition erfasst.

**Definition 3.7** *Sei $f = \langle Q_1 + \ldots + Q_m \rangle$ wobei $Q_1, \ldots, Q_m$ Minterme sind, und seien weiterhin $P_1, \ldots, P_n$ sämtliche Primimplikanten von $f$.*

1. *$P_i$ heisst* Kernimplikant *von $f$, wenn $\langle Q_j \rangle \rightarrow \langle P_i \rangle$, $\langle Q_j \rangle \not\rightarrow \langle P_k \rangle$ für ein beliebiges $j \in \{1, \ldots, n\}$ und alle $k \neq i$.*
2. *$P_i$ heisst* absolut eliminierbar*, wenn aus $\langle Q_j \rangle \rightarrow \langle P_i \rangle$ folgt $\langle Q_j \rangle \rightarrow \langle P_k \rangle$ für einen Kernimplikanten $P_k$.*

Den offensichtlichen Zusammenhang zwischen irredundanten Summen, Kernimplikanten und absolut eliminierbaren Primimplikanten formulieren wir dann als Lemma.

**Lemma 3.3** *Sei $P$ ein Primimplikant einer Schaltfunktion $f$.*

1. *$P$ ist ein Kernimplikant von $f$ genau dann, wenn $P$ in jeder irredundanten Summe von Primimplikanten für $f$ vorkommt.*
2. *$P$ ist ein absolut eliminierbarer Primimplikant von $f$ genau dann, wenn $P$ in keiner irredundanten Summe von Primimplikanten für $f$ vorkommt.*

**Beispiel 3.5** *Abschliessend betrachten wir noch ein weiteres Beispiel. Es wird wieder unmittelbar mit der Überdeckunstabelle begonnen (der Leser möge sich selbst davon überzeugen, dass die Primimplikanten korrekt sind).*

| | $\bar{d}e$ | $a\bar{b}d$ | $cd\bar{e}$ | $a\bar{b}e$ | $\bar{a}cd$ | $\bar{a}ce$ | $\bar{b}cd$ | $\bar{b}ce$ |
|---|---|---|---|---|---|---|---|---|
| $abcd\bar{e}$ | | | $\otimes$ | | | | | |
| $abc\bar{d}e$ | $\otimes$ | | | | | | | |
| $ab\bar{c}de$ | $\otimes$ | | | | | | | |
| $a\bar{b}cde$ | | $\times$ | | $\times$ | | | $\times$ | $\times$ |
| $a\bar{b}cd\bar{e}$ | | $\times$ | $\times$ | | | | $\times$ | |
| $a\bar{b}c\bar{d}e$ | $\times$ | | | $\times$ | | | | $\times$ |
| $a\bar{b}\bar{c}de$ | | $\times$ | | $\times$ | | | | |
| $a\bar{b}\bar{c}d\bar{e}$ | | $\otimes$ | | | | | | |
| $a\bar{b}\bar{c}de$ | $\times$ | | | $\times$ | | | | |
| $\bar{a}bcde$ | | | | | $\times$ | $\times$ | | |
| $\bar{a}bcd\bar{e}$ | | | $\times$ | | $\times$ | | | |
| $\bar{a}bc\bar{d}e$ | $\times$ | | | | | $\times$ | | |
| $\bar{a}b\bar{c}de$ | $\otimes$ | | | | | | | |
| $\bar{a}\bar{b}cde$ | | | | | $\times$ | $\times$ | $\times$ | $\times$ |
| $\bar{a}\bar{b}cd\bar{e}$ | | | $\times$ | | $\times$ | | $\times$ | |
| $\bar{a}\bar{b}c\bar{d}e$ | $\times$ | | | | | $\times$ | | $\times$ |
| $\bar{a}\bar{b}\bar{c}de$ | $\otimes$ | | | | | | | |

*Als Kernimplikanten erkennt man unmittelbar $\bar{d}e$, $a\bar{b}d$ und $cd\bar{e}$. Wenn man die davon überdeckten Zeilen weglässt, erhält man die folgende Überdeckungsmatrix.*

| | $\bar{a}cd$ | $\bar{a}ce$ | $\bar{b}cd$ | $\bar{b}ce$ |
|---|---|---|---|---|
| $\bar{a}bcde$ | $\times$ | $\times$ | | |
| $\bar{a}\bar{b}cde$ | $\times$ | $\times$ | $\times$ | $\times$ |

*Damit ergeben sich die folgenden irredundanten Summen von Primimplikanten:*

1. $\bar{d}e + a\bar{b}d + cd\bar{e} + \bar{a}cd$
2. $\bar{d}e + a\bar{b}d + cd\bar{e} + \bar{a}ce$

*Welche der beiden Realisierungen zu wählen ist, sollte im konkreten Fall anwendungsbezogen entschieden werden.*

## 3.4. KV-Diagramme

Die bisherigen Techniken zur Ermittlung von Primimplikanten und irredundanten Summen sind in ihrer Anwendung für den Menschen sehr ungeeignet, und die im vorherigen Abschnitt beschriebene Methode zur Ermittlung von minimalen Summen hat darüberhinaus den Nachteil, dass sie als Ausgangspunkt die Mintermnormalform benötigt.

In diesem Abschnitt wird die Ermittlung von Primimplikanten mit Hilfe von *KV-Diagrammen*, benannt nach dem amerikanischen Elektroingenieur Maurice Karnaugh

und dem amerikanischen Mathematiker Edard W. Veitch, dargestellt. Der Leser sollte bereits im Umgang mit KV-Diagrammen geübt sein, so dass diese hier nur kurz wiederholt werden.

**Definition 3.8 (Karnaugh-Veitch-Diagramme (KV-Diagramme))** *Ein* KV-Diagramm *ist eine tabellarische Darstellung einer $n$-stelligen Schaltfunktion $f \in F_n$, bei der allen Belegungen $(b_1, \ldots, b_n) \in \mathbb{B}^n$ ein Feld im KV-Diagramm zugeordnet ist. Die Felder sind dabei derart angeordnet, dass sich die Belegungen benachbarter Felder nur in exakt einem Wert $b_i$ mit $1 \leq i \leq n$ unterscheiden, und dass die Ränder des KV-Diagramms als toroidal verbunden gedacht werden. Die Felder werden nummiert mit der als Binärzahl interpretierten Belegung $(b_1, \ldots, b_n)$ der formalen Parameter der Funktion.*

Damit ist offensichtlich, dass jedes Feld in einem KV-Diagramm den Wert eines Minterms der dargestellten Funktion enthält. Die Position des Minterms in der Matrix bestimmt, welche Kombination der Eingangsvariablen (d.h. der Funktionsparameter) vorliegt. Betrachten wir dazu zunächst einige Beispiele für KV-Diagramme.

**Beispiel 3.6** *Sei $f_2 \in F_2$ die durch folgende Wertetabelle definierte 2-stellige Schaltfunktion. Das zugehörige KV-Diagramm für diese Funktion ist auf der rechten Seite dargestellt.*

| $a$ | $b$ | $f_2(a,b)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$f_2(a,b)$ :

| $a$ \ $b$ | | |
|---|---|---|
| | 0 (0) | 1 (1) |
| | 1 (2) | 1 (3) |

*Sei weiter $f_3 \in F_3$ die folgende 3-stellige Schaltfunktion, für die wieder auf der rechten Seite das zugehörige KV-Diagramm angegeben ist.*

| $a$ | $b$ | $c$ | $f_3(a,b,c)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

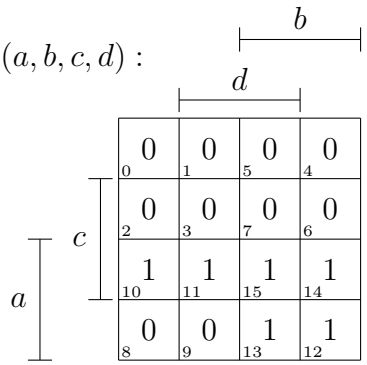
$f_3(a,b,c)$ :

| $b$ \ $a$,$c$ | | | | |
|---|---|---|---|---|
| | 0 (0) | 1 (1) | 0 (5) | 1 (4) |
| | 1 (2) | 0 (3) | 1 (7) | 0 (6) |

*Abschliessend betrachten wir noch die Funktion $f_4 \in F_4$ mit dem zugehörigen KV-Diagramm.*

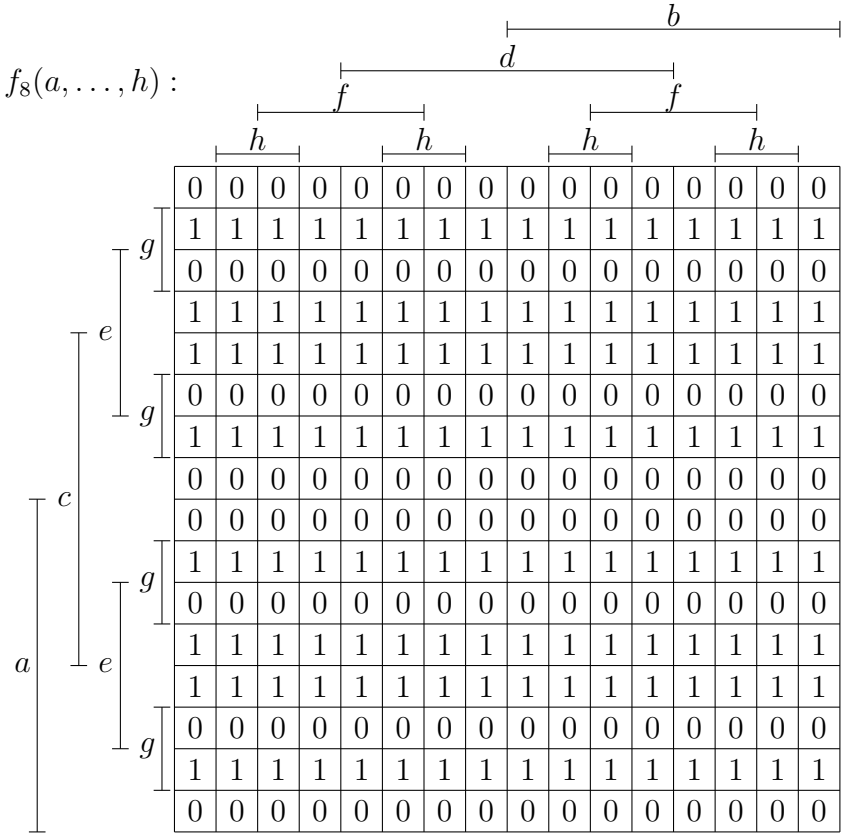
| $a$ | $b$ | $c$ | $d$ | $f_4(a,b,c,d)$ | $a$ | $b$ | $c$ | $d$ | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

$f_4(a,b,c,d)$ :

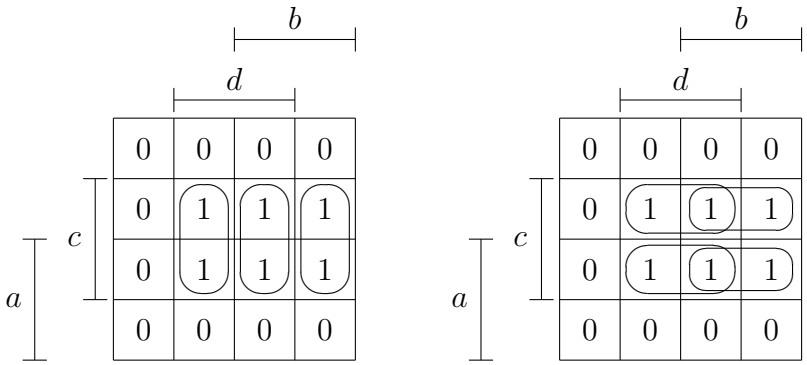



KV-Diagramme sind eine für den Menschen sehr übersichtliche und leicht zu verstehende Form der Darstellung von Schaltfunktionen. Allerdings sind auch KV-Diagramme in ihrer Nützlichkeit beschränkt auf Schaltfunktionen kleinerer Stelligkeit. Betrachten wir dazu als (abschreckendes) Beispiel das folgende KV-Diagramm einer 8-stelligen Schaltfunktion $f_8$.

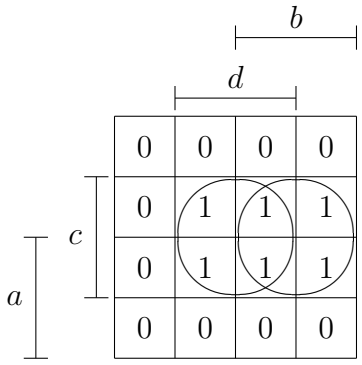
$f_8(a,\ldots,h)$ :




In einem KV-Diagramm repräsentiert jeder Block von $2^k$ Feldern, die mit dem Wert 1 belegt sind, einen *Implikanten* der dargestellten Funktion. Im folgenden sind alle aus 2 Blöcken bestehenden Implikanten der Schaltfunktion $f = \langle bc + cd \rangle$ dargestellt. Aus Gründen der Übersichtlichkeit wurde die Darstellung auf zwei KV-Diagramme verteilt.




Weiter repräsentiert jeder *maximale* Block von $2^k$ Feldern, welche den Wert 1 enthalten, in einem KV-Diagramm einen *Primimplikanten* der dargestellten Funktion. Betrachten wir dazu erneut die Schaltfunktion $f = \langle bc + cd \rangle$. Offensichtlich hat $f$ genau zwei Primimplikanten, nämlich $bc$ und $cd$.



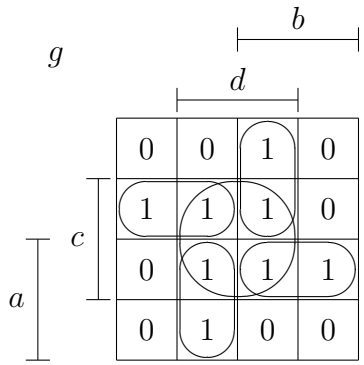

Die so gefundenen Primimplikanten lassen sich nun nach den bekannten Kriterien unterscheiden, um eine minimal Überdeckung mit Hilfe von KV-Diagrammen zu finden:

1. *Kernimplikanten* sind solche Implikanten, die mindestens ein 1-Feld allein überdecken.

2. *Absolute eliminierbare Primimplikanten* sind solche Implikanten, deren sämliche 1-Felder durch Kernimplikanten überdeckt werden.

3. *Teilweise redundante Primimplikanten* sind alle übrigen so erhaltenden Primimplikanten. Sie überdecken sich derart, dass einige von ihnen entfernt werden können, ohne die Überdeckung insgesamt zu verletzen.

Im vorherigen Beispiel der Schaltfunktion $f = \langle bc + cd \rangle$ handelt es demzufolge bei $bc$ und $cd$ um Kernimplikanten, da beide Primimplikanten 1-Felder überdecken, die von keinem anderen Primimplikanten überdeckt werden. Betrachten wir dazu zunächst noch ein weiteres Beispiel:




Bei $\bar{a}\bar{b}c$, $\bar{a}bd$, $abc$ und $a\bar{b}d$ handelt es sich um Kernimplikanten, da sie jeweils exakt ein 1-Feld überdecken, welches durch keinen anderen Primimplikanten überdeckt wird. Jeder dieser Kernimplikanten überdeckt darüberhinaus jeweils ein 1-Feld, welches auch durch den Primimplikanten $cd$ überdeckt wird. Folglich handelt es sich bei $cd$ um einen absolut eliminierbaren Primimplikanten.

Daraus lässt sich ein einfaches Verfahren zum Auffinden von irredundanten Summen mit Hilfe von KV-Diagrammen ableiten:

1. Markiere alle Primimplikanten im KV-Diagramm.
2. Wähle alle Kernimplikanten aus.
3. Streiche alle absolut eliminierbaren Primimplikanten.
4. Wähle eine Teilmenge der verbliebenen teilweise redundanten Primimplikanten, die alle nicht bereits durch Kernimplikanten überdeckten 1-Felder überdeckt und in einer möglichst geringen Anzahl von Literalen in der Darstellung als Schaltformel resultiert.

Dieses Verfahren scheint auf den ersten Blick sehr effizient zu sein, besitzt jedoch im Prinzip diesselbe Problemkomplexität wie die allgemeinen Minimierungsalgorithmen. Es profitiert im Wesentlichen von der Erkenntnis, dass die graphisch orientierte Darstellung dem Menschen entgegenkommt, der in der Regel zwar keine optimale, aber zumindest eine gute Lösung *sieht*.

Diese Fähigkeit nimmt jedoch mit steigender Variablenanzahl ab, wie bereits zuvor anhand eines KV-Diagramms mit 8 Variablen eindrucksvoll demonstriert worden ist. Mit steigender Variablenanzahl werden nämlich die Adjazenzbeziehungen zwischen benachbarten Feldern komplexer, und so verliert sich schnell der Vorteil der übersichtlichen Repräsentation interessierender Eigenschaften der dargestellten Funktion.

## 3.5. Positional Cube Notation

Wie beschrieben sind KV Diagramme für die manuelle Minimierung bis zu einer gewissen Variablenanzahl sehr gut geeignet. Für die Verarbeitung in einer Maschine sind

sie in ihrer allgemeinen Form jedoch völlig ungeeignet, da für die Darstellung bereits exponentieller Speicherbedarf notwendig ist. Aus den KV Diagrammen lässt sich jedoch eine einfache, effiziente Darstellung für Schaltfunktionen ableiten.

Hierzu stellen wir zunächst fest, dass ein KV Diagramm für eine $n$-stellige Schaltfunktion lediglich eine Projektion des $n$-dimensionalen Variablenraums in eine zweidimensionale Ebene ist. Jeder Knoten in diesem Raum repräsentiert einen Minterm und eine bestimmte Zusammenfassung von $2^k$ Knoten repräsentiert einen Implikanten. Eine derartige Zusammenfassung bezeichnet man als *Würfel* (engl.: *cube*). Ein *Würfel* wird streng formal als $n$-Tupel über 0, 1 und $X$ definiert, wobei 0, 1 und $X$ die üblichen Bedeutungen „Low“, „High“ und „Don't Care“ zukommen. Daraus lässt sich eine effiziente Repräsentation für Schaltfunktionen ableiten, die sogenannte *Cube Notation*.

**Definition 3.9 (Cube Notation)** *Sei $f \in F_n$ eine $n$-stellige Schaltfunktion und seien $Q_1, \ldots, Q_m \in F_{\{x_1,\ldots,x_n\}}$ Produktterme mit $f = \langle Q_1 + \ldots + Q_m \rangle$. Dann lässt sich jeder Implikant $Q_i$ durch ein $n$-Tupel $\vec{c}_i = (c_{i,0}, \ldots, c_{i,n})$ darstellen, wobei*

$$c_{i,j} \quad = \quad \begin{cases} 0, & \textit{falls } \bar{x}_j \in Q_i \\ 1, & \textit{falls } x_j \in Q_i \\ X, & \textit{sonst.} \end{cases}$$

*Die Schaltfunktion ist dann repräsentiert durch die Menge $C_f = \{\vec{c}_1, \ldots, \vec{c}_m\}$ der so gebildeten Tupel.*

Offensichtlich lässt sich aus der Cube Notation leicht die Repräsentation als Schaltformel erzeugen, so dass hier keine eigene Semantik für die Cube Notation definiert werden muss. Stattdessen legen wir fest, dass die Semantik der Cube Notation über die Semantik der zugehörigen Schaltformel definiert ist.

**Beispiel 3.7** *Sei $f \in F_4$ eine 4-stellige Schaltfunktion mit $f = \langle ab\bar{c} + \bar{b}c + cd \rangle$. Für die Implikanten der Funktionen ergeben sich die zugehörigen Quadrupel zu $\vec{c}_{ab\bar{c}} = (1, 1, 0, X)$, $\vec{c}_{\bar{b}c} = (X, 0, 1, X)$ und $\vec{c}_{cd} = (X, X, 1, 1)$. $f$ lässt sich dann durch*

$$C_f \quad = \quad \{(1, 1, 0, X), (X, 0, 1, X), (X, X, 1, 1)\}$$

*darstellen.*

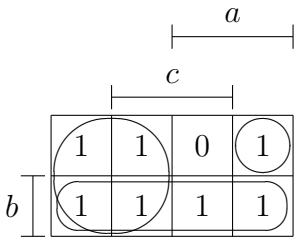
Eine für die Verarbeitung in der Maschine günstigere Darstellung von 0, 1 und $X$ in Würfeln ist diese als Paare von Binärzahlen zu kodieren, also 0 als 10, 1 als 01 und $X$ als 11 zu kodieren. Diese Darstellung ist so zu verstehen, dass 10 den ersten der beiden Werte (0) darstellt, 01 den zweiten Wert (1) und 11 beide möglichen Werte. Die Kodierung 00 entspricht ist ungültig und wird nicht benutzt zur Kodierung von Implikanten. Diese spezielle Kodierung der Cube Notation wird als *Positional Cube Notation* bezeichnet.

**Beispiel 3.8** *Betrachten wir erneut die Schaltfunktion $f = \langle ab\bar{c} + \bar{b}c + cd \rangle$ aus dem vorangegangenen Beispiel. Dann ergibt sich die folgende Darstellung in Positional Cube Notation:*

| *Implikant* | $\bar{a}a$ | $\bar{b}b$ | $\bar{c}c$ | $\bar{d}d$ |
|---|---|---|---|---|
| $ab\bar{c}$ | 01 | 01 | 10 | 11 |
| $\bar{b}c$ | 11 | 10 | 01 | 11 |
| $cd$ | 11 | 11 | 01 | 01 |

Diese Kodierung erlaubt es Operationen auf Schaltfunktionen in der Maschine sehr effizient als einfache Matrixoperationen zu realisieren. Beispielsweise lässt sich der Schnitt zweier Implikanten durch bitweise Multiplikation der Spaltenwerte implementieren. Betrachten wir dazu die Implikanten in folgender Tabelle (rechts ist zur Verdeutlichung das zugehörige KV-Diagramm dargestellt):

| Implikant | $\bar{a}a$ | $\bar{b}b$ | $\bar{c}c$ |
|---|---|---|---|
| $\bar{a}$ | 10 | 11 | 11 |
| $b$ | 11 | 01 | 11 |
| $a\bar{b}\bar{c}$ | 01 | 10 | 10 |




Um nun den Schnitt der beiden Implikanten $\bar{a}$ und $b$ zu bilden, genügt es die Bitwerte spaltenweise zu multiplizieren:

| | $\bar{a}a$ | $\bar{b}b$ | $\bar{c}c$ |
|---|---|---|---|
| $\bar{a}$ | 10 | 11 | 11 |
| $b$ | 11 | 01 | 11 |
| | 10 | 01 | 11 |

Der Implikant $\bar{a}b$ bildet demzufolge den Schnitt von $\bar{a}$ und $b$, was sich anhand des obigen KV-Diagramms leicht verifizieren lässt. Die Schnitte von $\bar{a}$ mit $a\bar{b}c$ und $b$ mit $a\bar{b}c$ sind leer, da die entstehenden Implikanten die ungültige Spaltenkodierung 00 enthalten. Das bedeutet, es existert kein Implikant, der gleichzeitig von $a\bar{b}c$ und $\bar{a}$ oder $b$ überdeckt wird, was sich anhand des KV-Diagramms wieder leicht überprüfen lässt.

## 3.6. Der ESPRESSO Algorithmus

Die bisher vorgestellten Verfahren sind für die praktische Anwendung in dieser Form nicht geeignet, da die Erzeugung aller Primimplikanten einer Schaltfunktion im allgemeinen exponentiellen Aufwand erfordert. Ursache hierfür ist, dass die Anzahl der Primimplikanten ungünstigstenfalls exponentiell proportional zur Anzahl der Eingangsvariablen ist. Das anschliessende Finden irredundanter Summen ist ein allgemeines Überdeckungsproblem, und als solches NP-vollständig, besitzt also ebenfalls exponentielle Laufzeit.

Gesucht ist demzufolge ein Verfahren, welches zu einer gegebenen Schaltfunktion nur eine geringe Menge von Primimplikanten erzeugt, und aus diesen eine irredundante Summe ableitet. Das Verfahren soll akzeptable Laufzeit besitzen unter Verzicht auf die optimale Lösung.

Ein solches Verfahren ist der 1986 von Richard Rudell in [Rud86] vorgestellte ESPRESSO Algorithmus. Hierbei handelt es sich um den wohl bekanntesten heuristischen Algorithmus zur Logikminimierung, der seit den achtziger Jahren in fast allen kommerziellen Softwareprodukten zum Einsatz kommt. Genaugenommen ist in [Rud86] eine Variante namens ESPRESSO-MV beschrieben, die die zuvor in ESPRESSO-I, ESPRESSO-II, ESPRESSO-IIC und EXPRESSO-EXACT für zweiwertige Logiken entwickelten Ansätze auf mehrwertige Logiken verallgemeinert. Die ESPRESSO-MV Variante hat sich heutzutage alle anderen Varianten ersetzt.

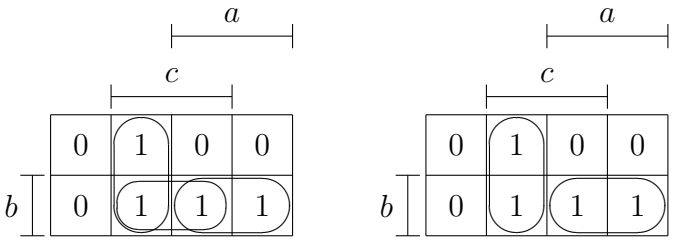
Der ESPRESSO Algorithmus besteht im Kern aus den drei Operatoren EXPAND, IRREDUNDANT und REDUCE, die im Folgenden detailliert beschrieben werden. Daneben wurden zur weiteren Optimierung des Verfahrens noch zusätzliche Operatoren eingeführt, von denen die interessantesten Operatoren ESSENTIALS und LAST_GASP ebenfalls kurz vorgestellt werden. Die Beschreibung der Operatoren basiert im wesentlichen auf den Erläuterungen aus [Rud86] und Lehrgangsunterlagen der Universität Oldenburg.

### 3.6.1. Der EXPAND Operator

Das Herzstück des ESPRESSO Algorithmus bildet der EXPAND Operator, indem alle Schritte zusammengefasst sind, die sich auf die *Vergrösserung* (engl.: *expansion*) bzw. das Zusammenfassen von Implikanten beziehen. Somit ist EXPAND für sich allein bereits ein Minimierungsalgorithmus: Der in [Bro81] beschriebene PRESTO Algorithmus bestand beispielsweise ausschliesslich aus diesem Operator. Für realistische Minimierungsprobleme ist jedoch die Wahrscheinlichkeit für nichtminimale Ergebnisse des EXPAND Operators zu gross, so dass Verbesserungen in Form zusätzlicher Operatoren erforderlich sind.

Der EXPAND Operator selbst ist wieder unterteilt in einzelne zu bearbeitende Schritte, die nachfolgend im Detail beschrieben werden.

**Schritt 1: Ordne alle Implikanten** Die exakte Minimierung durch Generieren *aller* Primimplikanten der Schaltfunktion mit anschliessender Auswahl der besten Lösung kommt aufgrund des zuvor geschilderten Laufzeitproblems nicht in Frage. Stattdessen soll mit Hilfe einer ausgeklügelten Heuristik auf *direktem* Weg minimiert werden. Dazu werden die Implikanten in einer bestimmten Reihenfolge expandiert.



Im KV-Diagramm auf der linken Seite wurde offensichtlich eine schlechtere Reihenfolge für die Expandierung gewählt als auf der rechten Seite, denn das Ergebnis enthält einen (überflüssigen) Implikanten mehr. Bei der für das rechte Diagramm gewählten Reihenfolge ist die Generierung des Implikanten $bc$ nicht erforderlich, da die Minterme $\bar{a}bc$ und $abc$ bereits durch die Primimplikanten $\bar{a}c$ und $ab$ überdeckt werden. Im ersten Diagramm müsste dieser Implikant nachträglich entfernt werden, da er offensichtlich redundant ist.

Insgesamt ist es wünschenswert keine Expandierungen durchzuführen, die sich im nachhinein als überflüssig herausstellen. Intuitiv lässt sich dieses Ziel leicht dadurch erreichen, dass zunächst die essentiellen Implikanten expandiert werden. Allerdings müssten hierfür wieder (zumindest temporär) alle Primimplikanten gebildet werden, was aber genau vermieden werden soll. Stattdessen soll die Lösung direkt gebildet werden, und es müssen demnach andere Daten zur Entscheidung, welche Implikanten expandiert werden sollen, herangezogen werden.

Hierzu kann eine einfache Heuristik verwendet werden. Die Idee dabei ist, dass es günstig ist, zunächst diejenigen Implikanten zu expandieren, die am unwahrscheinlichsten durch die Expansion anderer Implikanten überdeckt werden. Am unwahrscheinlichsten werden diejenigen überdeckt, die am wenigsten andere Implikanten *in der Nähe* haben.

Die *Nähe* zweier Implikanten ist dabei definiert als die Anzahl von Literalen, in denen diese Implikanten übereinstimmen. Für die Feststellung, wieviele Implikanten sich *in der Nähe* befinden, untersucht man also die einzelnen Literale eines Implikanten daraufhin, ob diese ingesamt – also in allen Implikanten – eher häufig oder weniger häufig vorkommen. Diese Berechnung kann basierend auf der Positional Cube Notation verhältnismässig einfach durchgeführt werden:

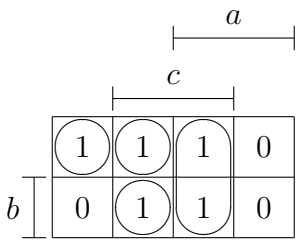
1. Zunächst wird die *Spaltensumme* für jede Spalte der Matrix gebildet, d.h. alle in einer Spalte vorkommenden 1 Werte werden aufaddiert.

2. Anschliessend werden die Spaltensummen der Spalten aufaddiert, in denen beim betrachteten Implikanten eine 1 steht. Dadurch erhält man einen Zahlenwert, der aussagt, wie oft die 1 Werte des Implikanten insgesamt in anderen Implikanten vorkommen.

Diese durch Addition der Spaltensummen berechnete Zahl wird als *Gewicht* des Implikanten bezeichnet. Je kleiner dieses Gewicht ist, desto seltener kommen die Werte des Implikanten in anderen Implikanten vor, und dementsprechend weniger Implikanten befinden sich in der Nähe. Betrachten wir dazu das folgende Beispiel:

| Implikant | $\bar{a}a$ | $\bar{b}b$ | $\bar{c}c$ | Gewicht |
|---:|---|---|---|---|
| $ac$ | 01 | 11 | 01 | $1+3+2+4=10$ |
| $\bar{a}bc$ | 10 | 01 | 01 | $3+2+4=9$ |
| $\bar{a}\bar{b}c$ | 10 | 10 | 01 | $3+3+4=10$ |
| $\bar{a}\bar{b}\bar{c}$ | 10 | 10 | 10 | $3+3+1=7$ |
| Summe: | 3 1 | 3 2 | 1 4 | |

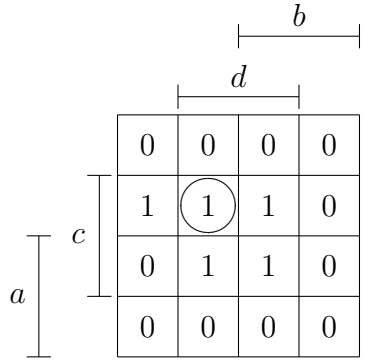
Die Implikanten werden nun gemäss ihrere Gewichte geordnet, Implikanten mit kleinerem Gewicht werden vor Implikanten mit grösserem Gewicht sortiert. Damit ist dann eine Reihenfolge erreicht, die wahrscheinlich eine erfolgreiche Expansion ermöglicht. Im Beispiel würde also zunächst versucht $\bar{a}\bar{b}\bar{c}$ zu expandieren, dann $\bar{a}bc$ und zuletzt $\bar{a}\bar{b}c$ und $ac$.




Anhand der Darstellung im KV-Diagramm lässt sich schon vermuten, dass diese Reihenfolge in einem EXPAND Schritt zur optimalen Lösung führt. Die Details der Expandierung werden nachfolgend beschrieben.

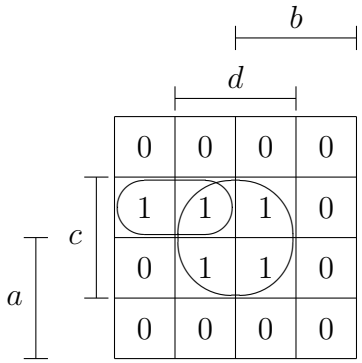
**Schritt 2: Expandiere jeden Implikanten** Die Implikanten werden nun in der zuvor bestimmten Reihenfolge nacheinander untersucht.

**Schritt 2.1: Überdecke andere Implikanten** Für jeden Implikanten wird nun geprüft, ob eine Expansion möglich ist, die weitere Implikanten überdeckt. Existieren dabei für einen Implikanten mehrere Möglichkeiten zu expandieren, so stellt sich die Frage, wie expandiert werden soll. Betrachten wir dazu zunächst wieder ein Beispiel: Im folgenden KV-Diagramm soll der markierte Implikant $\bar{a}\bar{b}cd$ expandiert werden. Welche der möglichen Expansionen soll gewählt werden?




Der EXPAND Operator führt zur Beantwortung dieser Frage zunächst temporär *alle* Expansionen eines Implikanen durch, indem er versucht, der Reihe nach Zusammenfassungen mit jedem anderen Implikanten durchführt. Es wird dann die Expansion ausgewählt, die am meisten Implikanten zusammenfasst. Sollten hier mehrere Expansionen in Frage kommen, so wird einfach die erste in der Liste gewählt.

Für das Beispiel bedeutet dies, dass zunächst eine Expansion des Implikanten zu $cd$ gewählt wird und anschliessend der Implikant $\bar{a}\bar{b}cd$ zu $\bar{a}\bar{b}c$ expandiert wird, wie nachfolgend dargestellt:




Diese Heuristik verfolgt das grundlegende Ziel, bei der Minimierung die Anzahl der Implikanten möglichst klein zu machen. In den meisten Fällen, wie in diesem Beispiel, ist es prinzipiell egal, welche Expansion zuerst ausgewählt wird, da die andere Auswahlmöglichkeit ohnehin in einem der Folgeschritte ausgewählt werden würde. Es gilt allerdings zu beachten, dass der Algorithmus in der Praxis nicht wie in diesem Beispiel mit 5 Implikanten arbeitet, sondern mit beispielsweise 500 Implikanten. Für die Anwendung ist es demzufolge schon entscheidend, ob hier die Anzahl der Implikanten im ersten Schritt auf 400 oder 200 reduziert wird (in Anlehnung daran, dass die Anzahl im Beispiel auf 2 oder 4 reduziert werden konnte). Im allgemeinen gilt, dass weniger Implikanten auch weniger Rechenzeit bedeuten.

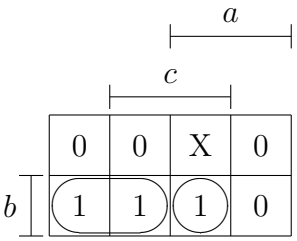
**Schritt 2.2: Expandiere weiter** Nach dem Zusammenfassen des gerade betrachteten Implikanten mit anderen Implikanten existeren eventuell noch immer Möglichkeiten, den Implikanten zu vergrössern bzw. zu expandieren, ohne mit dieser Expansion andere Implikanten zu überdecken.

Durch diese Expansion lässt sich zum einem die Anzahl der Literale weiter reduzieren. Zum anderen erhöht die weitere Expansion eines Implikanten die Wahrscheinlichkeit, dass er *näher* an mehr anderen Implikanten ist. Dies muss als Vorbereitung auf die möglicherweise folgende Anwendung des REDUCE Operators verstanden werden: Ein Implikant kann hier am wahrscheinlichsten erfolgreich *reduziert* werden, wenn sich möglichst viele andere Implikanten in seiner *Nähe* befinden.

Zur weiteren Expansion von Implikanten existieren offensichtlich zwei Möglichkeiten: Durch Schneiden von anderen Implikanten und durch Ausnutzen von *Don't Care*-Termen. Nachfolgend wird zunächst die Expansion durch Schneiden von Implikanten erläutert, und anschliessend die Expansion unter Ausnutzung von *Don't Care*-Termen.

**Schritt 2.2.1: Schneide andere Implikanten** Auf den ersten Blick scheint es egal zu sein, ob man im unten dargestellten Beispiel beim Expandieren von $abc$ den anderen

Implikanten schneidet oder den *Don't Care*-Term überdeckt, denn Anzahl und Grösse der Implikanten sind im Ergebnis identisch.




Versteht man allerdings wie gesagt, diese weitere Expansion von Implikanten als Vorbereitung für den REDUCE Operator, so ist die Zielsetzung, möglichst viele andere Implikanten zu schneiden, denn der Implikant soll ja möglichst *nah* an möglichst viele andere Implikanten heran. Für das vorangegangene Beispiel bedeutet dies, dass $abc$ mit $\bar{a}bc$ zusammengefasst wird.

**TODO:** Noch ein Beispiel mit mehreren Schnitten

**Schritt 2.2.2: Expandiere maximal** Nachdem in vorangegangenen Schritt sukzessive möglichst viele andere Implikanten geschnitten wurden, wird nun eventuell noch weiter expandiert und zwar unter Ausnutzung von *Don't Care*-Termen. Hierbei wird in ESPRESSO nun tatsächlich die grösstmögliche Expansion ermittelt.

Bis zu diesem Schritt wurde ein Implikant mit Hilfe von Heuristiken schon möglichst gut expandiert. D.h. das Problem den grösstmöglichen Primimplikanten zu finden wurde immer weiter vereinfacht, aber bis zu diesem Schritt nicht endgültig gelöst. Nun wird angenommen, dass das Problem hinreichend klein ist, um es effizient mit Hilfe von Standardverfahren zu lösen. Dazu wird es als Überdeckungsproblem formuliert und in optimaler Weise gelöst.

Es genügt hierbei zu verstehen, dass die maximale Expansion in diesem Schritt aufgrund der vorherigen Schritte wahrscheinlich effizienter durchgeführt werden kann, da voraussichtlich nicht mehr so viele mögliche Expansionen möglich sind und diese deshalb *exakt* untersucht werden können, ohne Rückgriff auf ein heuristisches Verfahren.

### Bewertung des Ergebnisses

Aufgrund der eingesetzten Heuristiken gibt es, wie bereits zuvor erwähnt, keine Garantie, dass das von EXPAND erreichte Ergebnis minimal ist, d.h. dass die erzielte Primimplikantenanzahl die kleinstmögliche ist.

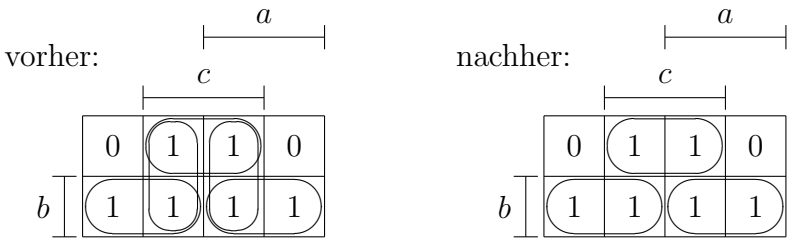
Bei guten Heuristiken liefert EXPAND schon sehr gute Ergebnisse. Für eine hinreichend geringe Variablenanzahl existieren auch Heuristiken, die garantiert eine minimale Lösung liefern. In diesen Fällen sind dann auch einfache Minimierer, wie der bereits angesprochene PRESTO Algorithmus (vgl. [Bro81]), gut geeignet für die Optimierung von

Schaltungen. Bei größeren Designproblemen muss das Ergebnis von EXPAND allerdings verbessert werden.

Bei diesen nicht optimalen Ergebnissen lassen sich zwei Fälle unterscheiden, die beim ESPRESSO Algorithmus jeweils durch die Operatoren IRREDUNDANT und REDUCE behandelt werden, und im Folgenden näher untersucht werden.

## 3.6.2. Der IRREDUNDANT Operator

Der IRREDUNDANT Operator behandelt schlechte – d.h. nicht minimale – Ergebnisse des EXPAND Operators, bei denen Implikanten der erzeugten Lösungsmenge redundant bzw. überflüssig sind. IRREDUNDANT optimiert diese Ergebnisse durch Entfernen dieser Redundanzen, wie im folgenden Beispiel angedeutet:




Man unterscheidet dabei zwei Arten von Redundanzen: *Totale Redundanzen* und *partielle Redundanzen*. Ein Implikant wird als total redundant bezeichnet, wenn er vollständig von einem oder mehreren anderen Implikanten überdeckt wird. Zwei oder mehr Implikanten werden als partiell redundant bezeichnet, wenn jeder dieser Implikanten für sich entfernt werden kann, jedoch nicht alle Implikanten gleichzeitig.

Das Problem bei partieller Redundanz ist, die größtmögliche Anzahl partiell redundanter Implikanten zu löschen, so dass die übrig bleibende, irredundante Implikantenmenge immer noch die zugrunde liegende Schaltfunktion repräsentiert.

Zum Entfernen aller Redundanzen geht der IRREDUNDANT Operator in den nachfolgend beschriebenen Schritten vor.

**Schritt 1: Bestimme alle (total und partiell) redundanten Implikanten** Es werden zunächst *alle* Implikanten als (potentiell) redundant markiert. Dann werden diejenigen Implikanten in die Lösung übernommen, die jeweils als einziges einen oder mehrere Minterme überdecken, d.h. die Kernimplikanten. Alle übrigen Implikanten sind total oder partiell redundant.

**Schritt 2: Lösche alle total redundanten Implikanten** Aus der Lösung werden alle total redundanten Implikanten entfernt, d.h. alle Implikanten, die vollständig von einem oder mehreren anderen Implikanten überdeckt werden. Alle nun noch (eventuell vorhandenen) potentiell redundanten Implikanten müssen partiell redundant sein.

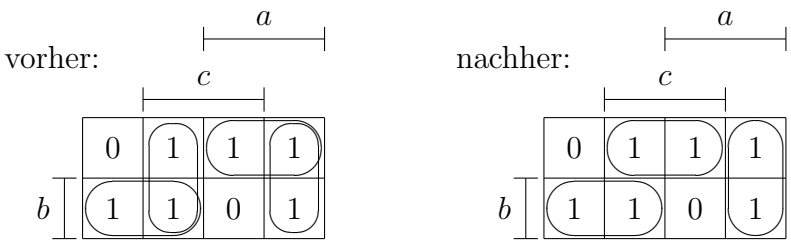
**Schritt 3: Übernehme einen partiell redundanten Implikanten** Ziel ist es, die grösstmögliche Anzahl partiell redundanter Implikanten zu löschen, so dass die übrig bleibende, irredundante Implikantenmenge immer noch die zugrunde liegende Schaltfunktion spezifiert. Dies entspricht wieder dem bereits in Schritt 2.2.2 des EXPAND Operators aufgetretenen Überdeckungsproblem. Mit Hilfe eines Standardlösungsverfahrens für dieses Problem könnte in einem Schritt die maximale Anzahl partiell redundanter Implikanten entfernt werden.

Statt eines exakten Verfahrens bedient man sich an dieser Stelle jedoch wieder einer Heuristik. Dabei übernimmt man denjenigen partiell redundanten Implikanten, welcher am meisten noch nicht von den bisher übernommenen Implikanten überdeckte Minterme überdeckt. Sollte hier mehr als ein Implikant in Frage kommen, wird wieder der erste in Frage kommende ausgewählt.

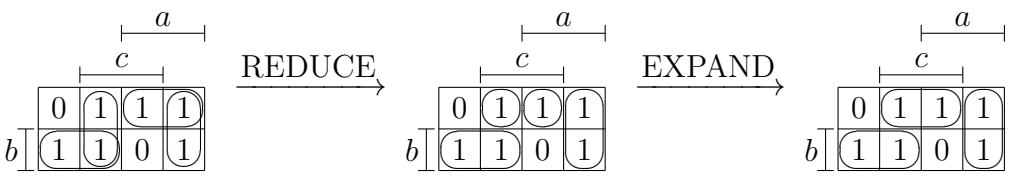
Durch das Übernehmen eines partiell redundanten Implikanten sind eventuell weitere Implikanten total redundant geworden, weshalb nun Schritt 2 wiederholt wird, um diese total redundanten Implikanten zu entfernen. Anschliessend wird wieder heuristisch versucht, den nächsten partiell redundanten Implikanten zu übernehmen, bis alle potentiell redundanten Implikanten entweder als total redundant entfernt oder als partiell redundant übernommen wurden, und die verbliebene Implikantenmenge somit irredundant ist.

## 3.6.3. Der REDUCE Operator

Der REDUCE Operator behandelt nicht minimale Ergebnisse des EXPAND Operators, die auf unglückliche Entscheidungen der verwendeten Heuristiken zurückzuführen sind, welche ihrerseits dazu führten, dass Implikanten nicht optimal zusammengefasst wurden. Der REDUCE Operator hat eine Verbesserung dieser Ergebnisse zum Ziel, d.h. eine Verringerung bzw. Reduzierung der Anzahl von Primimplikanten, was im folgenden Beispiel veranschaulicht wird:




Wie leicht zu ersehen ist, lässt sich nur dann eine kleinere Implikantenmenge erzeugen, wenn zuvor bestimmte Zusammenfassungen wieder aufgelöst werden. Genau dieses Auflösen von zuvor zusammengefassten Implikanten ist die Aufgabe des REDUCE Operators. Nach REDUCE erfolgt ein weiterer Aufruf des EXPAND Operators, so dass eventuell neue, kostengünstigere Zusammenfassungen gebildet werden können. Dieser Verlauf ist nachfolgend dargestellt:

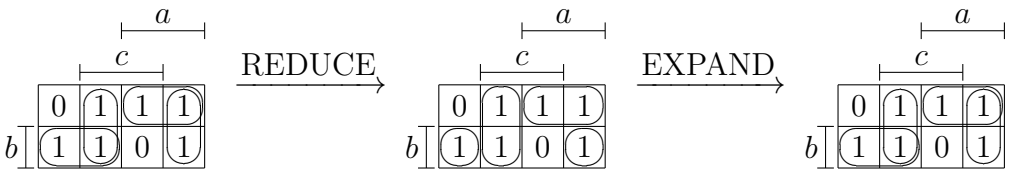



Hierbei soll die Anzahl der Implikanten im Verlauf der Reduzierung auch temporär nicht vergrössert werden, sondern konstant bleiben. Diese Einschränkung ist sehr wichtig und erfüllt zwei Aufgaben:

1. Durch eine REDUCE-EXPAND Folge wird nie ein *schlechteres* Ergebnis als vorher erreicht, sondern immer ein mindestens genauso gutes oder besseres Ergebnis, d.h. ein ungünstiges Auflösen von zusammengefassten Implikanten hat nicht zur Folge, dass EXPAND nun mehr Primimplikanten als vorher bildet.

2. Die Implikantenmenge wird auch temporär nicht größer: Dies würde der Grundidee der Verminderung von Speicher- und Rechenzeitbedarf beim ESPRESSO Algorithmus widersprechen. Bei einer temporären Erhöhung der Anzahl, z.B. von 2 auf 4 Implikanten lässt sich zwar kein großartig erhöhter Speicherbedarf erkennen, eine Übertragung des Sachverhalts auf eine Anzahl von 200 und 400 Implikanten sollte das Problem jedoch verdeutlichen.

Analog zum EXPAND Operator ist auch das Ergebnis des REDUCE Operators davon abhängig, in welcher Reihenfolge die Reduzierung der Implikanten versucht wird. Die folgende Abbildung zeigt dasselbe Beispiel wie oben, jedoch mit einer anderen Reduktionsreihenfolge, so dass in diesem Fall keine Verbesserung erzielt werden kann:




Durch die Einschränkung, dass die Anzahl der Implikanten konstant bleiben muss, kann nur eine bestimmte Anzahl Implikanten überhaupt reduziert werden, so dass die erfolgversprechensten Implikanten zuerst bearbeitet werden sollten, d.h. die Implikanten müssen geordnet werden.

Der erfolgversprechenste Implikant für die Reduzierung ist gerade derjenige Implikant, der am wahrscheinlichsten durch die Expansion anderer Implikanten überdeckt wird. Am wahrscheinlichsten wird aber gerade der Implikant überdeckt, welcher am meisten andere Implikanten *in der Nähe* hat. Die verwendete Heuristik entspricht damit genau einer Umkehrung der Heuristik, die für den EXPAND Operator verwendet wird. Es wird also nicht zunächst der Implikant mit dem kleinsten, sondern derjenige mit dem größten Gewicht betrachtet.

Im Unterschied zur bei EXPAND verwendeten Heuristik bezieht sich die hier verwendete Heuristik nur auf einen Implikanten. Es wird also nur der zuerst zu reduzierende Implikant durch diese Heuristik ermittelt, die restlichen Implikanten werden anders geordnet. Begründet wird dieser Ansatz durch die Feststellung, dass die nach dem zuerst reduzierten Implikanten betrachteten Implikanten zum Ergebnis der Reduktion des ersten Implikanten *passen* müssen.

Es ist dann nämlich nicht wichtig, dass diese Implikanten ein hohes Gewicht haben, sondern dass sie sich möglichst *nah* an der ersten Reduktion befinden. Nur so besteht die Möglichkeit, dass ihre Reduktion sich mit der Reduktion des ersten Implikanten in einem folgenden EXPAND Schritt zusammenfassen lässt. Dementsprechend werden nach der Reduktion des ersten Implikanten die übrigen Implikanten nach ihrer *Ähnlichkeit* zum reduzierten Implikanten geordnet. Die Ähnlichkeit zweier Implikanten definiert sich als die Anzahl der Spaltenwerte der Positional Cube Notation, in denen sie übereinstimmen. Diese Operation lässt sich entsprechend effizient durch Zählen der Übereinstimmungen implementieren.

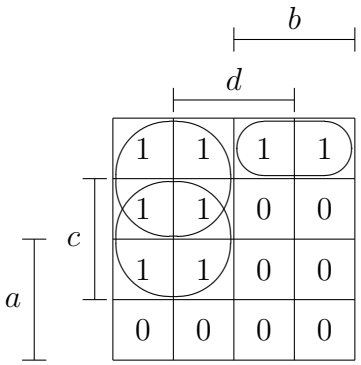
| Implikant | $\bar{a}a$ | $\bar{b}b$ | $\bar{c}c$ | Gewicht | Ähnlichkeit |
|---|---|---|---|---|---|
| $ac$ | 01 | 11 | 01 | 10 | |
| $\bar{a}\bar{b}c$ | 10 | 10 | 01 | 10 | 3 |
| $\bar{a}bc$ | 10 | 01 | 01 | 9 | 3 |
| $\bar{a}\bar{b}\bar{c}$ | 10 | 10 | 10 | 7 | 1 |

Im Beispiel wurde $ac$ als zuerst zu reduzierender Implikant ausgewählt, da er neben $\bar{a}\bar{b}c$ mit 10 das höchste Gewicht hat[1], und die Ähnlichkeiten der übrigen Implikanten zu $ac$ durch spaltenweisen Vergleich berechnet.

In der durch Gewichtung und Ähnlichkeiten ermittelten Reihenfolge werden die Implikanten nun nacheinander zu reduzieren versucht. Sollte sich der durch Gewichtung zuerst ermittelte Implikant nicht reduzieren lassen (wegen temporärer Erhöhung der Implikantenanzahl), wird stattdessen versucht den gemäß der Gewichtungsheuristik reihenfolgenächsten Implikanten zu reduzieren. Die Ordnung der restlichen Implikanten erfolgt aus Gründen der Effizienz erst nach erfolgreicher Reduktion eines ersten Implikanten.

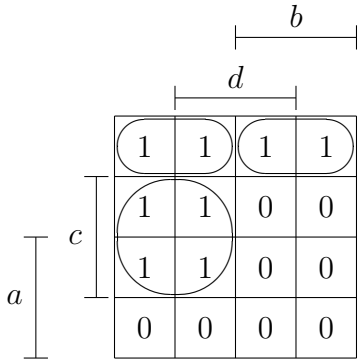
Für die Reduktion eines einzelnen Implikanten werden zunächst alle Minterme ermittelt, die nur durch diesen Implikanten überdeckt werden. Diese Minterme bezeichnet man als *essentielle Minterme*. Anschliessend wird versucht den kleinsten Implikanten zu bilden, der alle essentiellen Minterme überdeckt. Ist der Implikant nun kleiner als zuvor, so war die Reduktion erfolgreich. Ist er genauso groß wie vorher, so existiert kein kleinerer Implikant, welcher alle essentiellen Minterme umfaßt, und der Implikant kann somit nicht reduziert werden.

[1] Die Auswahl bei gleicher Gewichtung erfolgt wieder zufällig.




| | $\bar{a}a$ | $\bar{b}b$ | $\bar{c}c$ | $\bar{d}d$ |
|---|---|---|---|---|
| $\bar{a}b\bar{c}$ | 10 | 01 | 10 | 11 |
| $\bar{a}\bar{b}$ | 10 | 10 | 11 | 11 |
| $\bar{b}c$ | 11 | 10 | 01 | 11 |

In diesem Beispiel ergeben sich für die Implikanten die Gewichte $g_{\bar{a}\bar{b}} = 15$, $g_{\bar{b}c} = 14$ und $g_{\bar{a}b\bar{c}} = 12$. Entsprechend wird der Implikant $\bar{a}\bar{b}$ reduziert auf einen Implikanten, welcher nur noch die essentiellen Minterme $\bar{a}\bar{b}\bar{c}\bar{d}$ und $\bar{a}\bar{b}\bar{c}d$ überdeckt, also auf $\bar{a}\bar{b}\bar{c}$. Weitere Reduktionen sind nicht mehr möglich, da alle verbliebenen Implikanten nur noch essentielle Minterme überdecken. Damit erhalten wir für den REDUCE Schritt folgendes Ergebnis:




In einem darauf folgenden EXPAND Schritt werden nun hoffentlich die Implikanten $\bar{a}\bar{b}\bar{c}$ und $\bar{a}b\bar{c}$ zusammengefasst, und somit die minimale Realisierung der Schaltungsfunktion ermittelt.

### 3.6.4. Iterative Verbesserungsstrategie

Wie in den vorangegangenen Abschnitten deutlich wurde, werden möglicherweise *schlechte*, also nicht minimale Ergebnisse des EXPAND Operators durch eine nachfolgende Anwendung der Operatoren IRREDUNDANT und REDUCE korrigiert. Der Ablauf des ESPRESSO Algorithmus lässt sich also durch das in Abbildung 3.1 dargestellte Flussdiagramm beschreiben.
Da nach dem zweiten Aufruf von EXPAND wieder die Möglichkeit von Redundanzen besteht, erfolgt ebenfalls ein weiterer Aufruf von IRREDUNDANT. Falls die Anzahl Primimplikanten nach dem zweiten Aufruf von IRREDUNDANT ebenso groß ist, wie vor dem Aufruf von REDUCE, so konnte offenbar durch REDUCE keine Verbesserung

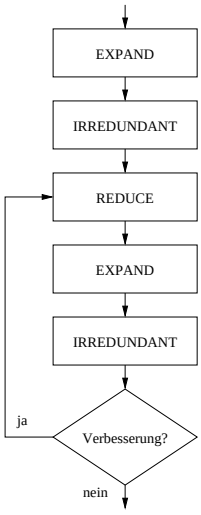



Abbildung 3.1.: Der ESPRESSO Algorithmus

erreicht werden. Ein erneuter Aufruf von REDUCE wäre dann *wahrscheinlich* nicht erfolgversprechend. Hat sich die Anzahl Primimplikanten jedoch verringert, so konnte REDUCE offenbar eine Verbesserung des Ergebnisses erreichen. Eventuell existiert dann die Möglichkeit einer weiteren Reduktion, denn die Primimplikantenmenge hat sich verändert. Ein erneuter Aufruf von REDUCE ist demnach sinnvoll. Dieses Grundkonzept des ESPRESSO Algorithmus wird als *iterative Verbesserungsstrategie* bezeichnet.

Die Entscheidung, ob eine weitere Iteration erfolgen soll, beruht auf einer Heuristik: Wurde in einem Durchlauf keine Verbesserung erzielt, so ist es *wahrscheinlich*, dass eine weitere Iteration ebenfalls keine Verbesserung erreichen kann. Dies ist jedoch nur dann mit Sicherheit der Fall, wenn die neue Lösung exakt der vorherigen Lösung entspricht. Handelt es sich um eine andere Lösung mit derselben Anzahl von Implikanten, so ist nicht sicher, ob nicht doch in einem folgenden Iterationsschritt noch eine Verbesserung möglich wäre.

Da dieser Fall jedoch *sehr unwahrscheinlich* ist und sich die Abfrage am Ende der Iterationsschleife wesentlich komplizierter gestalten würde (man müsste hier vergleichen, ob die neue Lösung exakt der vorangegangenen Lösung entspricht, also zum einen die alte Lösung speichern und zum anderen eine möglicherweise große Anzahl Implikanten paarweise miteinander vergleichen), lautet die Strategie einfach, die Iterationsschleife an dieser Stelle abzubrechen.

Nach Verlassen der Schleife lässt sich immer noch nicht garantieren, dass die so erreichte Lösung minimal ist. Zum einen wegen der zuvor erwähnten geringen, aber dennoch vorhandenen Wahrscheinlichkeit, dass eine erneute Iteration doch eine Verbesserung zur Folge hätte, und zum anderen aufgrund der Tatsache, dass auch der REDUCE Operator mit Heuristiken arbeitet, so dass zum Beispiel bei gleicher Gewichtung von Implikan-

ten der Listenerste für die nächste Reduktion bestimmt wird, d.h. die Ergebnisqualität dieser Reduktion eher zufällig ist.

### 3.6.5. Erweiterungen

Neben den in den vorangegangenen Abschnitten beschriebenen Operatoren, die das Grundgerüst des ESPRESSO Algorithmus ausmachen, existieren noch einige Erweiterungen, die eine zusätzliche Steigerung der Effizienz des Algorithmus zum Ziel haben. Von diesen Erweiterungen wollen wir an dieser Stelle den ESSENTIALS und den LAST_GASP Operator betrachten.

#### Der LAST_GASP Operator

Bei der Betrachtung des Kernalgorithmus im vorherigen Abschnitt wurde bereits erläutert, dass sich nach Beendigung der Hauptschleife nicht garantieren lässt, dass das erreichte Ergebnis minimal ist. Um die Wahrscheinlichkeit für die Existenz einer besseren Lösung noch weiter zu reduzieren, wird nach dem letzten Schleifendurchlauf ein weiterer Operator angewendet: der sogenannte LAST_GASP Operator (*last gasp: „ein letztes Mal nach Luft schnappen“*).

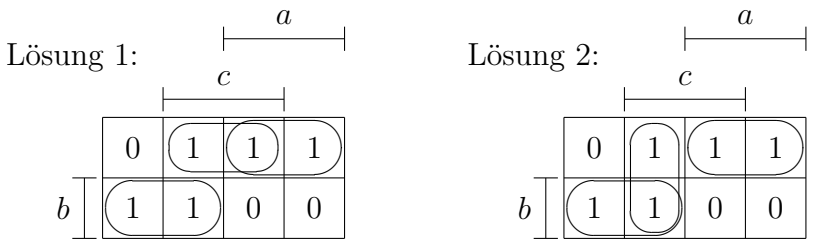
Eventuell konnte nämlich keine Verbesserung mehr erzielt werden, weil die Heuristiken von REDUCE oder EXPAND ungünstige Entscheidungen getroffen haben. Der LAST_GASP Operator führt deshalb eine weitere, vollständige Verbesserungsiteration unter Verwendung *anderer* Heuristiken als in der Hauptschleife durch. Falls sich die REDUCE und EXPAND Heuristiken in einer Art *Sackgasse* befunden haben, so besteht hierbei eine große Wahrscheinlichkeit dafür, dass diese *Sackgasse* durch Verwendung anderer Heuristiken überwunden werden kann.

Mit anderen Worten, die Wahrscheinlichkeit, dass zwei verschiedene Heuristiken beide am gleichen Problem scheitern, ist verschwindend gering. Falls also doch noch eine Verbesserung der Lösung möglich ist, so wird sie mit hoher Wahrscheinlichkeit vom LAST_GASP Operator gefunden. Im Fall einer Verbesserung besteht dann wieder die Möglichkeit weiterer Verbesserungen durch eine REDUCE, EXPAND und IRREDUNDANT Iteration. Es wird also wieder an den Anfang der Hauptschleife verzweigt.

#### Der ESSENTIALS Operator

Es existeren Implikanten, die in jeder möglichen Lösung enthalten sein müssen, da sie als einzige einen oder mehrere Minterme der zu spezifizierenden Schaltfunktion überdecken. Im Rahmen der Erklärungen zum IRREDUNDANT Operator tauchten schon sogenannte *essentielle* Implikanten auf, die ebenfalls als einzige einen oder mehrere Minterme überdecken und somit aus der Menge der von IRREDUNDANT untersuchten potentiell redundanten Implikanten entfernt werden können. Dort handelt es sich um Implikanten, die in der aktuell betrachteten *lokalen* Lösung enthalten sein müssen.

Hier aber sind solche Implikanten gemeint, die in *jeder* Lösung, also sozusagen *global* enthalten sein müssen, also eine spezielle Art von Primimplikanten, nämlich *Kernimplikanten*. Da diese global essentiellen Implikanten in jeder möglichen Lösung enthalten sein müssen, werden sie durch die Hauptschleife des ESPRESSO Algorithmus nicht verändert. Selbst bei einer Reduktion würde EXPAND diese Implikanten wiederherstellen.




Im Beispiel sieht man dann dass die beiden Kernimplikanten $\bar{a}b$ und $a\bar{b}$ in jeder möglichen Lösung enthalten sein müssen. Durch den ESSENTIALS Operator werden genau diese global essentiellen Implikanten *vor* dem Aufruf der Hauptschleife bestimmt und gekennzeichnet, wodurch die Menge der in jedem Schleifendurchlauf zu berücksichtigenden Implikanten verkleinert werden kann, was bei größeren Problemen eine effizientere Verarbeitung ermöglicht.

### 3.6.6. Zusammenfassung

Der ESPRESSO Algorithmus mitsamt der vorgestellten Erweiterungen ist in Abbildung 3.2 dargestellt. Abschliessend werden die einzelnen Operatoren nocheinmal kurz zusammengefasst:

- EXPAND

  Ermittelt Primimplikanten durch Zusammenfassen, Schneiden und Vergrößern.

- IRREDUNDANT

  Entfernt redundante Primimplikanten.

- ESSENTIALS

  Kennzeichnet global essentielle Primimplikanten – d.h. Kernimplikanten – zwecks effizienterer Iteration.

- REDUCE

  Löst Zusammenfassungen wieder auf, um in einem nachfolgenden EXPAND Schritt ggfs. bessere Zusammenfassungen ermitteln zu können.

- LAST_GASP

  Führt eine Iteration mittels anderer Heuristiken durch zwecks Auflösung von *Sackgassen*, die durch die EXPAND und REDUCE Heuristiken nicht überwunden werden konnten.

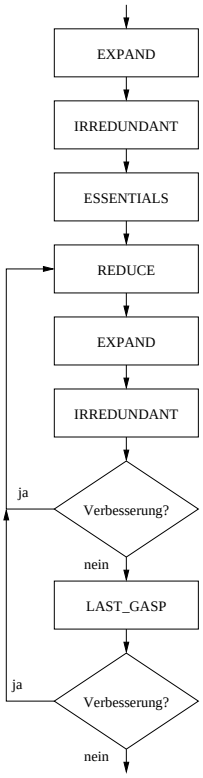



Abbildung 3.2.: Der erweiterte ESPRESSO Algorithmus

### 3.6.7. Fazit

ESPRESSO ist nach wie vor der am meisten benutzte Algorithmus für die 2-stufige Logikmimierung, denn ESPRESSO ist extrem schnell und sehr robust. Ausserdem ist der Algorithmus, aufgrund seiner einfachen Basisstruktur, leicht durch Optimierungen zu ergänzen (wie die zuvor vorgestellten Operatoren ESSENTIALS und LAST_GASP demonstrieren).

Üblicherweise werden weniger als 5 EXPAND/REDUCE/IRREDUNDANT-Durchläufe benötigt. In den meisten Fällen sind sogar nur 1-2 Durchläufe notwendig. Das erreichte Ergebnis ist dabei in fast allen Fällen nur wenige Prozent schlechter als die optimale Lösung.

Beispielsweise konnte bereits 1984 mit Hilfe von ESPRESSO eine Schaltformel mit 3172 Termen und 23741 Literalen in weniger als 16 Sekunden minimiert werden, auf einem Prozessor, der rund 10 Millionen Instruktionen pro Sekunde verarbeiten konnte. In heutigen Werkzeugen und auf modernen Rechnern werden „derart kleine Probleme“ in Bruchteilen von Sekunden minimiert. Dies liegt nicht zuletzt daran, dass die verwendete Datenstruktur – die Positional Cube Notation – eine sehr effiziente Speicherung und Verarbeitung durch die Maschine ermöglicht.

# 4. Model Checking

## 4.1. Validierung von Systemen

Der Begriff der *Validierung von Systemen* beschreibt den Prozess zur Überprüfung der Korrektheit von Spezifikationen, Entwürfen und Produkten in der Informationsverarbeitung. Dieser Prozeß gewinnt zunehmend an Bedeutung. Seit Ende der 60er Jahre des vergangenen Jahrhunderts wurden deshalb unterschiedliche Ansätze entwickelt, um die Korrektheit von Systemen nachzuweisen oder zumindest die Fehlerwahrscheinlichkeit zu reduzieren.

Da jedoch generell nicht alle Fehler vermieden werden können, wurden *Simulation* und *Testing* als Techniken zum Aufspüren von Fehlern entwickelt. Die Simulation ermöglicht das Nachweisen von Fehlern in der Designphase, das Testing kann nur auf das fertige Produkt oder Modul angewandt werden. Während diese Methoden leicht anzuwenden sind, auch für unerfahrene Entwickler, und darüber hinaus viele Fehler relativ schnell und kostengünstig aufdecken, können sie aber weder Fehlerfreiheit garantieren, noch gibt es eine verlässliche Möglichkeit, die Anzahl der verbliebenen Fehler einzuschätzen.

Eine erfolgversprechendere Alternative stellt die *formale Verifikation* dar. Hierbei wird mittels einer geeigneten Logik die Korrektheit des betrachteten (Teil-)Systems bewiesen. Besonders im Bereich der Entwicklung digitaler Schaltungen haben formale Verifikationsverfahren in den letzten Jahren ein hohes Maß an Reife und Effizienz erreicht und finden zunehmend Einzug in die industrielle Entwurfspraxis.

## 4.2. Formale Verifikation

Heute existieren unterschiedliche Ansätze zur *formalen Verifikation* von Systemen, die grob nach folgenden Kriterien klassifiziert werden können (vgl. [HR04, S. 172f]):

- **Beweisbasiert vs. modellbasiert.** In einem beweisbasierten Verfahren wird das System als Menge von Formeln $\Gamma$ (in einer geeigneten Logik) und die Spezifikation als Formel $\phi$ beschrieben. Die Verifikation erfolgt dann als Auffinden eines Beweises für $\Gamma \models \phi$.
  In einem modellbasierten Verfahren wird stattdessen das System als Modell $\mathcal{M}_\mathcal{S}$ dargestellt, und die Spezifikation wiederum als eine Formel $\phi$. Die Verifikation beschränkt sich dann darauf zu prüfen, ob $\mathcal{M}_\mathcal{S} \models \phi$ gilt. Modellbasierte Verfahren sind üblicherweise einfacher, da sie auf einem einzigen Modell $\mathcal{M}_\mathcal{S}$ basieren,

während zum Beweis von $\Gamma \models \phi$ für alle Modelle $\mathcal{M}$ geprüft werden muss, ob $(\forall \psi \in \Gamma : \mathcal{M} \models \psi) \Rightarrow \mathcal{M} \models \phi$ gilt.

- **Grad der Automatisierung.** Die Verfahren unterscheiden sich darüber hinaus im Grad der Automatisierung. Die meisten Verfahren sind semiautomatisch, erfordern also immer noch einen gewissen Grad an Interaktion mit dem Designer.
- **Vollständige vs. teilweise Verifikation.** Die Spezifikationen können entweder das System als Ganzes beschreiben oder nur einen Teilbereich des Systems oder eine bestimmte interessante Eigenschaft. Üblicherweise will man nur bestimmte Eigenschaften verifizieren.
- **Anwendungsbereich,** in dem das Verfahren eingesetzt werden kann.
- **Phase der Entwicklung,** in der das Verfahren zum Einsatz kommt. Besonders interessant sind Verfahren, die schon in einer frühen Phase der Entwicklung zum Einsatz kommen, da die Kosten für Fehlerkorrekturen in frühen Phasen deutlich niedriger sind als in späteren Phasen.

Nach den Maßstäben der obigen Klassifizierung ist die in diesem Dokument vorgestellte Methode des *Model Checking* als automatisches, modellbasiertes Verfahren zum Beweisen von Eigenschaften eines nebenläufigen, reaktiven Systems einzuordnen. Kritische Fehler in nebenläufigen, reaktiven Systemen sind üblicherweise durch Testing schwer oder gar nicht zu finden, da sie selten einfach zu reproduzieren sind.

Eine für reaktive Systeme geeignete Logik stellt die *Linear Temporal Logic* (LTL) – auch als *Propositional Linear Temporal Logic* (PLTL) bezeichnet – dar, die im folgenden Abschnitt vorgestellt wird. Neben der LTL existieren noch weitere für Model Checking geeignete Logiken, zum Beispiel die *Computational Tree Logic* (CTL), die im Anschluss an die LTL vorgestellt wird.

## 4.3. Linear Time Logic

Dieser Abschnitt beschreibt kurz die Linear Time Logic. Ausführliche Einführungen in die Thematik finden sich unter anderem in [HR04] und [Kat99].

### 4.3.1. Syntax von LTL

Wie in der Aussagenlogik (vgl. Abschnitt 1.2) bilden *Propositionalzeichen*, also Aussagen, welche nicht weiter zerlegt werden können, die Grundlage für die Logik. Die Menge der Propositionalzeichen wird mit $\mathcal{AP}$ bezeichnet. Hinter einem Propositionalzeichen kann sich zum Beispiel eine Aussagen wie „$x$ ist größer als 0“ oder „$x$ ist gleich 1“ verbergen.

**Definition 4.1 (Syntax von LTL)** *Die Menge $\mathcal{L}_{LTL}$ aller gültigen LTL-Formeln ist induktiv wie folgt definiert:*

1. *Jedes Propositionalzeichen $p \in \mathcal{AP}$ ist eine Formel.*
2. *Sind $\phi, \psi \in \mathcal{L}_{LTL}$ Formeln, so sind auch $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$ und $\phi \rightarrow \psi$ Formeln.*
3. *Sind $\phi, \psi \in \mathcal{L}_{LTL}$ Formeln, so sind auch $\mathbf{X}\,\phi$, $\phi\,\mathbf{U}\,\psi$, $\phi\,\mathbf{R}\,\psi$, $\mathbf{F}\,\phi$ und $\mathbf{G}\,\phi$ Formeln.*

Offensichtlich ist die Menge der aussagenlogischen Formeln eine (echte) Teilmenge der Menge der LTL-Formeln. Hinzu kommen die temporalen Operatoren **X** („neXt“), **U** („Until“), **R** („Release“), **F** („Future“) und **G** („Globally“).

### 4.3.2. Semantik von LTL

Analog zur Aussagenlogik handelt es sich bei LTL-Formeln lediglich um syntaktische Objekte ohne weitere inhaltliche Bedeutung. Es gilt nun diesen syntaktischen Objekten eine Bedeutung in Form eines Wahrheitswertes zuzuordnen. Im Gegensatz zur Aussagenlogik lässt sich die Gültigkeit einer LTL-Formel aber nicht unmittelbar aus der Belegung der Propositionalzeichen bestimmen, sondern es ist darüberhinaus der zeitliche Verlauf des Systems zu beachten. Entsprechend muss die Definition der Semantik auch diesen zeitlichen Verlauf berücksichtigen.

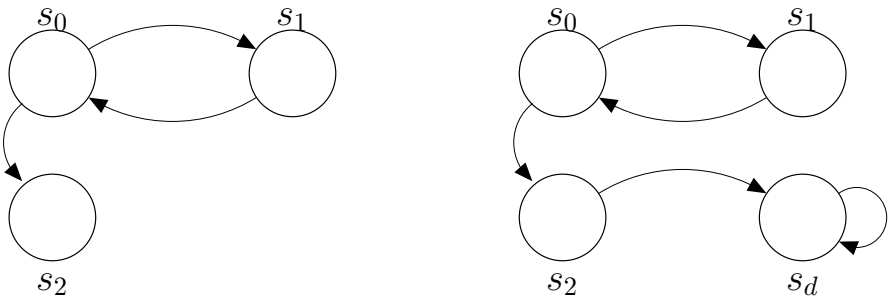
**Definition 4.2 (Transitionssystem)** *Ein* Transitionssystem *ist definiert als ein Triple $\mathcal{M} = (S, \rightarrow, l)$. Hierbei ist*

- *$S$ eine nicht-leere Menge von Zuständen,*
- *$\rightarrow \subseteq S \times S$ eine binäre Relation, so dass zu jedem Zustand $s \in S$ mindestens ein Folgezustand $s' \in S$ mit $s \rightarrow s'$ existiert, und*
- *$l : S \rightarrow \wp(P)$ eine „labelling function“, die jedem Zustand aus $S$ eine Menge von Propositionalzeichen zuordnet.*

Ein solches Transistionssystem wird auch häufig als *Kripke-Struktur* bezeichnet, da Kripke ähnliche Strukturen benutzte, um die Semantik für modale Logiken zu definieren [Kri63], wobei Kripke-Strukturen zusätzlich über eine ausgewiesene Menge $S_0 \subseteq S$ mit $S_0 \neq \emptyset$ von Startzuständen verfügen müssen.

Die Forderung, dass in einem Transitionssystem zu jedem Zustand $s \in S$ mindestens ein Folgezustand $s' \in S$ existieren muss, bedeutet, dass derartige Systeme nicht in Zuständen *stecken bleiben* können (engl.: *deadlock*). Hierbei handelt es sich in Wirklichkeit um eine technische Vereinfachung, denn jedes System, welches mindestens einen Zustand enthält, der stecken bleiben würde, kann stets um einen neuen Zustand $s_d \notin S$ erweitert werden, der den *Deadlock* repräsentiert, wie im nachfolgend dargestellten Beispiel illustriert.

**Beispiel 4.1 (Deadlocks)** *Betrachten wir den Graphen[1] des Systems auf der linken Seite. Dieses entspricht nicht der Definition eines Transitionssystems, da der Zustand $s_2$ keinen Folgezustand besitzt.*




*Erweitern wir das System nun um einen Zustand $s_d$, wie auf der rechten Seite gezeigt, und die Relation $\rightarrow$ um Paare $(s, s_d)$ für alle Zustände $s$, die zuvor steckengeblieben wären, sowie ein zusätzliches Paar $(s_d, s_d)$, so erhalten wir ein gültiges Transitionssystem. Intuitiv entspricht nun das Erreichen des Zustands $s_d$ dem „Deadlock" im ursprünglichen System.*

**Definition 4.3 (Pfad)** *Sei $\mathcal{M} = (S, \rightarrow, l)$ ein Transitionssystem. Ein* Pfad *$\pi$ in $\mathcal{M}$ ist eine unendliche Folge von Zuständen $s_1, s_2, s_3, \ldots \in S$ mit $s_i \rightarrow s_{i+1}$ für alle $i \geq 1$. Wir schreiben dann $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ für den Pfad. Die Menge aller Pfade über $S$ bezeichnen wir mit $S^\omega$.*

Ein Pfad $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ stellt eine mögliche Zukunft eines Transitionssystems dar: Zuerst befindet sich das Transitionssystem im Zustand $s_1$, anschließend im Zustand $s_2$ und so weiter. Wir schreiben $\pi^i$ für den Restpfad beginnend im Zustand $s_i$. Beispielsweise schreiben wir $\pi^4$ für den Pfad $s_4 \rightarrow s_5 \rightarrow \ldots$. Weiterhin schreiben wir $\pi(i)$ für den $i$-ten Zustand im Pfad $\pi$.

**Definition 4.4 (LTL-Modell)** *Ein Transitionssystem $\mathcal{M} = (S, \rightarrow, l)$ heisst* LTL-Modell, *wenn zu jedem Zustand $s \in S$ genau ein Folgezustand $s' \in S$ existiert.*

Statt LTL-Modell benutzen wir im weiteren Verlauf dieses Abschnitts den kürzeren Begriff *Modell*, da klar ist, dass es sich um ein LTL-Modell handelt.

**Definition 4.5 (Semantik von LTL)** *Sei $\mathcal{M} = (S, \rightarrow, l)$ ein Modell und sei weiter $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ ein Pfad in $\mathcal{M}$. Ob eine gegebene Formel durch den Pfad $\pi$ erfüllt wird, ist durch die nachfolgend definierte Erfüllbarkeitsrelation $\models$ bestimmt:*

1. *$\pi \models p$ gdw. $p \in l(s_1)$*
2. *$\pi \models \neg\phi$ gdw. $\pi \not\models \phi$*

[1] Zur Vereinfachung wurden hierbei die Markierungen für die Zustände weggelassen.

3. $\pi \models \phi \vee \psi$ *gdw.* $\pi \models \phi$ *oder* $\pi \models \psi$

4. $\pi \models \phi \wedge \psi$ *gdw.* $\pi \models \phi$ *und* $\pi \models \psi$

5. $\pi \models \phi \rightarrow \psi$ *gdw.* $\pi \not\models \phi$ *oder* $\pi \models \psi$

6. $\pi \models \mathbf{X}\,\phi$ *gdw.* $\pi^2 \models \phi$

7. $\pi \models \phi\,\mathbf{U}\,\psi$ *gdw. es existiert ein* $i \geq 1$ *mit* $\pi^i \models \psi$ *und* $\pi^j \models \phi$ *für alle* $1 \leq j < i$

8. $\pi \models \phi\,\mathbf{R}\,\psi$ *gdw. entweder ex.* $i \geq 1$ *so dass* $\pi^i \models \phi$ *und* $\pi^j \models \psi$ *für alle* $1 \leq j \leq i$, *oder für alle* $k \geq 1$ *gilt* $\pi^k \models \psi$

9. $\pi \models \mathbf{F}\,\phi$ *gdw. es existiert ein* $i \geq 1$ *mit* $\pi^i \models \phi$

10. $\pi \models \mathbf{G}\,\phi$ *gdw.* $\pi^i \models \phi$ *für alle* $i \geq 1$

Die Punkte *1.* bis *5.* stimmen mit der bereits bekannten Semantik der Aussagenlogik überein. Für **X**-Formeln ist dann lediglich zu prüfen, ob $\phi$ im nächsten Zustand gilt, also auf dem Restpfad, welcher den aktuellen Zustand nicht enthält.

Von den übrigen temporalen Operatoren ist **U**, als Abkürzung für „Until“, der wohl am häufigsten verwendete. Die LTL Formel $\phi\,\mathbf{U}\,\psi$ gilt auf einem Pfad, wenn $\phi$ zumindest solange gilt, bis ein Zustand erreicht ist, indem $\psi$ gilt. Wichtig hierbei ist zu bemerken, dass $\phi\,\mathbf{U}\,\psi$ fordert, dass ein Zustand existieren muss, indem $\psi$ gilt. D.h. es reicht nicht aus, wenn $\phi$ auf ewig gilt, aber $\psi$ niemals gilt[2].

$\phi\,\mathbf{R}\,\psi$ wird als „Release“ bezeichnet, da $\psi$ solange gelten muss, bis ein Zustand erreicht ist, indem $\phi$ gilt (wobei $\psi$ auch in diesem Zustand noch gültig sein muss). $\psi$ wird also durch $\phi$ „befreit“. $\mathbf{F}\,\phi$ bedeutet, dass es in der Zukunft zumindest einen Zustand geben muss, der $\phi$ erfüllt, während $\mathbf{G}\,\phi$ fordert, dass alle zukünftigen Zustände $\phi$ erfüllen müssen.

**Definition 4.6** *Sei* $\mathcal{M} = (S, \rightarrow, l)$ *ein Modell,* $s \in S$ *ein Zustand in* $\mathcal{M}$ *und* $\phi \in \mathcal{L}_{LTL}$ *eine LTL-Formel. Wir schreiben* $\mathcal{M}, s \models \phi$, *falls für jeden in* $s$ *beginnenden Pfad* $\pi$ *von* $\mathcal{M}$ *gilt:* $\pi \models \phi$.

Falls $\mathcal{M}$ aus dem Zusammenhang klar ist, schreiben wir kurz $s \models \phi$ statt $\mathcal{M}, s \models \phi$. Man sagt in diesem Fall: $\phi$ *gilt in* $s$ oder $s$ *erfüllt* $\phi$.

**Definition 4.7 (Semantische Äquivalenz)** *Zwei LTL-Formeln* $\phi, \psi \in \mathcal{L}_{LTL}$ *heissen* semantisch äquivalent, *oder kurz* äquivalent, *geschrieben* $\phi \equiv \psi$, *wenn für alle Modelle* $\mathcal{M}$ *und alle Pfade* $\pi$ *in* $\mathcal{M}$ *gilt:* $\pi \models \phi$ *gdw.* $\pi \models \psi$.

[2]Neben $\phi\,\mathbf{U}\,\psi$ wird deshalb häufig ein sogenanntes „Weak-until“ $\phi\,\mathbf{W}\,\psi$ angegeben, dessen Semantik sich nur darin von $\phi\,\mathbf{U}\,\psi$ unterscheidet, dass die Formel auch dann wahr ist, wenn $\phi$ ewig und $\psi$ niemals gilt.

Sind zwei Formeln $\phi$ und $\psi$ (semantisch) äquivalent, so sagt man auch: $\phi$ und $\psi$ sind *semantisch austauschbar*. Darunter versteht man die folgende Eigenschaft: Wenn $\phi$ Teilformel einer größeren Formel $\chi$ ist, und $\phi \equiv \psi$ gilt, so kann jedes Vorkommen von $\phi$ in $\chi$ durch $\psi$ ersetzt werden, ohne dass die Bedeutung der Formel $\chi$ verändert wird. Dies entspricht der bereits aus der Aussagenlogik bekannten Äquivalenz (vgl. Definition 1.11).

Aus der Aussagenlogik ist bereits bekannt, dass $\wedge$ und $\vee$ dual zueinander sind, es gelten also die folgenden Äquivalenzen:

$$\begin{aligned} \neg(\phi \vee \psi) &\equiv (\neg\phi) \wedge (\neg\psi) \\ \neg(\phi \wedge \psi) &\equiv (\neg\phi) \vee (\neg\psi) \end{aligned}$$

In einem ähnlichen Sinne sind auch jeweils $\mathbf{F}$ und $\mathbf{G}$ sowie $\mathbf{U}$ und $\mathbf{R}$ dual zueinander, und $\mathbf{X}$ ist dual zu sich selbst:

$$\begin{aligned} \neg\mathbf{G}\,\phi &\equiv \mathbf{F}\,\neg\phi \\ \neg\mathbf{F}\,\phi &\equiv \mathbf{G}\,\neg\phi \\ \neg(\phi\,\mathbf{U}\,\psi) &\equiv (\neg\phi)\,\mathbf{R}\,(\neg\psi) \\ \neg(\phi\,\mathbf{R}\,\psi) &\equiv (\neg\phi)\,\mathbf{U}\,(\neg\psi) \\ \neg\mathbf{X}\,\phi &\equiv \mathbf{X}\,\neg\phi \end{aligned}$$

Weiterhin gilt, dass $\mathbf{F}$ distributiv bezüglich $\vee$ und $\mathbf{G}$ distributiv bezüglich $\wedge$ ist:

$$\begin{aligned} \mathbf{F}\,(\phi \vee \psi) &\equiv \mathbf{F}\,\phi \vee \mathbf{F}\,\psi \\ \mathbf{G}\,(\phi \wedge \psi) &\equiv \mathbf{G}\,\phi \wedge \mathbf{G}\,\psi \end{aligned}$$

Es sei dem Leser überlassen zu zeigen, dass diese Äquivalenzen gelten. Die Beweise sind nahezu trivial, sollten aber als Übung zum Verständnis der LTL Semantik nicht unterbewertet werden. Der Leser sollte sich insbesondere klar machen, dass die folgenden beiden Äquivalenzen gelten:

$$\begin{aligned} \mathbf{F}\,\phi &\equiv \top\,\mathbf{U}\,\phi \\ \mathbf{G}\,\phi &\equiv \bot\,\mathbf{R}\,\phi \end{aligned}$$

Wie dem interessierten Leser sicherlich nicht entgangen sein wird, enthält die Menge $\mathcal{L}_{LTL}$ in einem gewissen Sinne zu viele Formeln. Ähnlich wie in der Aussagenlogik existieren Teilmengen von $\mathcal{L}_{LTL}$, welche bereits vollständig im Sinne der Semantik von LTL sind.

**Definition 4.8 (Adäquate Mengen)** *Sei $L \subseteq \mathcal{L}_{LTL}$. $L$ heisst* adäquat*, wenn für alle $\phi \in \mathcal{L}_{LTL}$ ein $\psi \in L$ existiert, so dass gilt: $\phi \equiv \psi$.*

Eine Menge $L$ von LTL-Formeln ist also genau dann *adäquat*, wenn sich für jede beliebige LTL-Formel $\phi$ eine Formel $\psi \in L$ finden lässt, welche die gleiche Bedeutung wie $\phi$ hat, für die also gilt: $\phi \equiv \psi$.

**Beispiel 4.2** *Sei $L \subseteq \mathcal{L}_{LTL}$ wie folgt induktiv definiert:*

1. *$p \in L$ für jedes $p \in \mathcal{AP}$.*

2. *Wenn $\phi, \psi \in L$, dann auch $\neg\phi \in L$, $\phi \wedge \psi \in L$, $\mathbf{X}\,\phi \in L$ und $\phi\,\mathbf{U}\,\psi \in L$.*

*Zeigen Sie, dass L eine* adäquate Menge *von LTL-Formeln ist. Existieren auch* adäquate Mengen*, die ausschliesslich aus Formeln bestehen, welche keine Vorkommen des* **X**-*Operators enthalten?*

## 4.3.3. LTL Spezifikationen

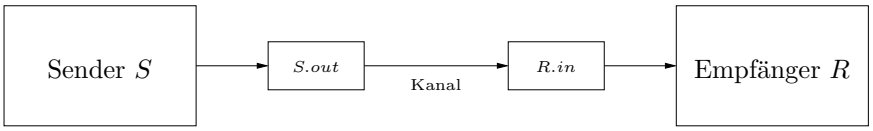
Um dem Leser eine bessere Vorstellung davon zu geben, auf welche Weise informale Eigenschaften in LTL formuliert werden können, werden zunächst zwei Beispiele behandelt, bevor dann der LTL-Model-Checking-Algorithmus vorgestellt wird. Die Beispiele entstammen [Kat99, S. 62ff]; im ersten Beispiel werden wir versuchen Eigenschaften eines einfachen Kommunikationssystems zu formalisieren, und im zweiten Beispiel werden formale Eigenschaften eines *Leader Election Protocols* spezifiziert.

### Kommunikationskanal

Wir betrachten einen unidirektionalen Kanal zwischen zwei kommunizierenden Entitäten: Einem Sender ($S$) und einem Empfänger ($R$). $S$ verfügt über einen Ausgabepuffer ($S.out$) und $R$ über einen Eingabepuffer ($R.in$). Beide Puffer haben nur eine endliche Kapazität. Wenn $S$ eine Nachricht $m$ an $R$ sendet, wird $m$ zunächst in den Ausgabepuffer $S.out$ kopiert. Der Ausgabepuffer $S.out$ und der Eingabepuffer $R.in$ sind über einen unidirektionalen Nachrichtenkanal verbunden. $R$ empfängt Nachrichten, indem sie aus dem Puffer $R.in$ entfernt werden. Wir nehmen an alle Nachrichten seien eindeutig identifizert und die Menge der Propositionalzeichen sei

$$\mathcal{AP} = \{m \in S.out \,|\, m \text{ ist Nachricht}\} \cup \{m \in R.in \,|\, m \text{ ist Nachricht}\}.$$

Darüber hinaus nehmen wir ebenfalls an, dass die Puffer $S.out$ und $R.in$ Nachrichten weder modifizieren noch verlieren, und dass Nachrichten auch nicht unendlich lange in einem Puffer verweilen.




Wir formalisieren die folgenden informalen Anforderungen an den Kommunikationskanal in LTL:

- „Eine Nachricht kann sich nicht zeitgleich in beiden Puffern befinden“:

$$\mathbf{G}\,\neg(m \in S.out \wedge m \in R.in)$$

- „Der Kanal verliert keine Nachrichten“. Mit anderen Worten, jede Nachricht, die sich jemals in $S.out$ befunden hat, wird sich auch irgendwann in $R.in$ befinden:

$$\mathbf{G}\left[m \in S.out \rightarrow \mathbf{F}\,(m \in R.in)\right]$$

  (Würden wir nicht voraussetzen, dass Nachrichten eindeutig identifiziert sind, so würde diese Eigenschaft nicht ausreichen, denn dann wäre die Eigenschaft auch erfüllt, wenn zwei Kopien einer Nachricht versendet werden, wovon nur die letzte empfangen wird. Verschiedene Autoren haben gezeigt, dass Eindeutigkeit von Nachrichten eine elementare Voraussetzung für die Spezifikation von Eigenschaften von nachrichten-basierten Systemen in Temporallogik ist). Die Gültigkeit der vorherigen Aussage vorausgesetzt, können wir nun ebenfalls

$$\mathbf{G}\left[m \in S.out \rightarrow \mathbf{X}\,\mathbf{F}\,(m \in R.in)\right]$$

  formulieren, denn $m$ kann niemals zeitgleich in $S.out$ und $R.in$ sein.

- „Der Kanal erhält die Reihenfolge der Nachrichten“. Dies bedeutet, wenn $S$ zunächst $m$ und später $m'$ sendet, dass $R$ auch zunächst $m$ und erst später $m'$ empfängt:

$$\begin{aligned}&\mathbf{G}\left[m \in S.out \wedge \neg m' \in S.out \wedge \mathbf{F}\,(m' \in S.out)\right.\\ &\left.\rightarrow \mathbf{F}\,(m \in R.in \wedge \neg m' \in R.in \wedge \mathbf{F}\,(m' \in R.in))\right]\end{aligned}$$

  Die Prämisse $\neg m' \in S.out$ wird benötigt, um festzulegen, dass sich $m'$ erst zu einem späteren Zeitpunkt als $m$ in $S.out$ befindet. $\mathbf{F}\,(m' \in S.out)$ alleine würde nicht ausschliessen, dass sich $m'$ bereits zu dem Zeitpunkt in $S.out$ befindet, zu dem $m$ in den Ausgabepuffer geschrieben wird.

- „Der Kanal generiert keine Nachrichten“. Diese Eigenschaft mag zunächst trivial erscheinen, ist aber je nach Realisierung des Kanals mit unter sehr schwierig sicher zu stellen. Sie besagt, dass jede Nachricht $m$ in $R.in$ sich zu einem beliebigen Zeitpunkt in der Vergangenheit einmal in $S.out$ befunden haben muss:

$$\mathbf{G}\left[(\neg m \in R.in)\,\mathbf{U}\,(m \in S.out)\right]$$

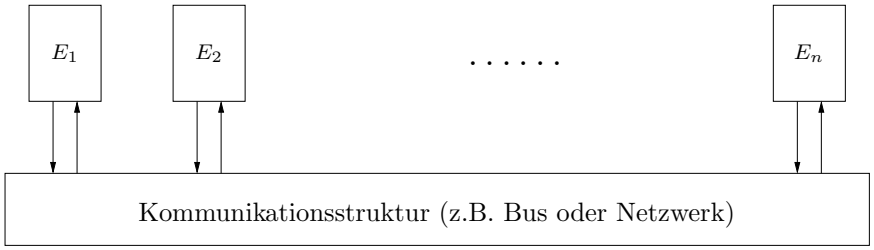
Bei den obigen Eigenschaften wird jeweils davon ausgegangen, dass implizit über alle Nachrichten quantifiziert wird (was ohne weiteres möglich ist, wenn man eine endliche Menge von Nachrichten voraussetzt).

### Dynamic Leader Election

In heutigen verteilten Systemen – unabhängig davon, ob es sich hierbei im Detail um Hardware- oder Softwaresysteme handelt – werden Funktionalität und Dienste von bestimmten Entitäten innerhalb des Systems zur Verfügung gestellt. Man denke dabei

zum Beispiel an Adresszuweisung und -registrierung, Koordination der Abfrageverarbeitung in verteilten Datenverwaltungssystemen, Uhrzeitabgleich, Initiierung von Topologieänderungen in einem Mobilfunknetzwerk, Load Balancing, etc. Üblicherweise sind mehrere Entitäten innerhalb des Systems in der Lage die Aufgabe zu erledigen, allerdings darf aus Gründen der Konsistenzsicherung immer nur eine Entität – bezeichnet als *„Leader“* – die Aufgabe durchführen. Unter den möglichen Leadern muss dann einer ausgewählt werden, was teilweise zufällig geschehen kann, häufig aber nach bestimmten Kriterien erfolgt (z.B. wähle die Entität, welche die besten Voraussetzung zur Lösung der Aufgabe mitbringt). In diesem Beispiel abstrahieren wir von speziellen Kriterien und verwenden stattdessen eine Bewertung, die jeder Entität eine Nummer zuordnet. Je höher die Nummer, desto stärker die Priorisierung.

Angenommen, es existiert eine endliche Anzahl von Entitäten, welche über eine beliebige Kommunikationsstruktur miteinander verbunden sind. Wie im vorangegangenen Beispiel gehen wir davon aus, dass die Kommunikation asynchron erfolgt.




Jede Entität hat eine eindeutige Nummer, oder auch *Identität*, und zur Vereinfachung wird angenommen, dass eine totale Ordnung auf Basis dieser Identitäten existiert. Das Verhalten der Entitäten ist in sofern *dynamisch*, dass sie initial inaktiv sind (also nicht an der Wahl teilnehmen) und zu beliebigen Zeitpunkten aktiv werden können (also an der Wahl teilnehmen). Um sicher zu stellen, dass überhaupt etwas passiert, wird angenommen, dass die Entitäten nicht ewig inaktiv sein können, d.h. jede Entität wird irgendwann einmal aktiv, und kann anschliessend auch nicht mehr inaktiv werden. Aus einer gegebenen Menge aktiver Entitäten wird dann ein Leader ausgewählt, und jedesmal wenn eine weitere inaktive Entität aktiv wird, findet eine neue Wahl statt, sofern diese neue Entität eine höhere Identität als der aktuelle Leader besitzt.

Die Menge der Propositionalzeichen sei

$$\mathcal{AP} = \{leader_i \,|\, 0 < i \leq n\} \cup \{active_i \,|\, 0 < i \leq n\} \cup \{i \sqsubset j \,|\, 0 < i, j \leq n\},$$

hierbei bedeutet $leader_i$, dass die $i$-te Entität Leader ist, $active_i$, dass die $i$-te Entität aktiv ist, und $i \sqsubset j$, dass die Identität von $i$ kleiner ist als die Identität von $j$ (im Sinne der totalen Ordnung). Im Folgenden benutzen wir $i, j$ als Identitäten. $n$ ist die Anzahl der Entitäten. Wir setzen voraus, dass niemals eine inaktive Entität Leader sein kann.

In der nachfolgenden Beschreibung der Eigenschaften werden wir All- und Existenzquantoren verwenden, die genaugenommen nicht Bestandteil der LTL sind. Aber da nur

eine endliche Menge betrachtet wird können wir $\forall i : \phi_i$ als syntaktischen Zucker für $\phi_1 \wedge \ldots \wedge \phi_n$ auffassen; analog für $\exists i : \phi_i$.

- „Es existiert stets ein Leader“:

$$\mathbf{G}\left[\exists i : leader_i \wedge (\forall j : j \neq i \rightarrow \neg leader_j)\right]$$

  Da wir jedoch ein dynamisches System beschreiben, indem all Entitäten zunächst inaktiv sein können (und somit kein Leader existieren kann), wird diese Eigenschaft im allgemeinen nicht erfüllt sein. Weiterhin ist zu beachten, dass aufgrund der asynchronen Kommunikation, der Wechsel von einem Leader zu einem anderen üblicherweise nicht atomar geschehen kann (d.h. es existiert ein Zeitfenster während dem Wechsel, in dem kein Leader existiert, oder gar zwei Leader gleichzeitig existieren). Unter diesen Gesichtspunkten sollte die obige Formel wie folgt geändert werden:

$$\mathbf{G}\,\mathbf{F}\left[\exists i : leader_i \wedge (\forall j : j \neq i \rightarrow \neg leader_j)\right]$$

  Dadurch wird es möglich, dass temporär kein Leader oder mehr als ein Leader existieren kann. Letzteres ist jedoch aus Gründen der Konsistenzsicherung selten erwünscht, entsprechend formulieren die nachfolgenden zwei Eigenschaften.

- „Es kann nur höchstens ein Leader existieren“:

$$\mathbf{G}\left[\forall i : leader_i \rightarrow (\forall j : j \neq i \rightarrow \neg leader_j)\right]$$

- „In jedem Zeitraum gibt es hinreichend viele Leader“. (Diese Anforderung verhindert die Konstruktion eines Auswahlverfahrens, welches niemals einen Leader wählt. Ein derartiges Protokoll würde immer noch die vorangegangene Eigenschaft erfüllen, wäre aber selbstverständlich nicht wünschenswert.)

$$\mathbf{G}\,\mathbf{F}\left[\exists i : leader_i\right]$$

  Diese Eigenschaft impliziert nicht, dass es unendlich viele Leader geben muss, sondern spezifiziert lediglich, dass es unendlich viele Zeitpunkte gibt, an denen ein Leader existiert.

- „Der aktive Leader wird zurücktreten, wenn eine Entität mit höherer Identität existiert“:

$$\mathbf{G}\left[\forall i, j : (leader_i \wedge i \sqsubset j \wedge \neg leader_j \wedge active_j) \rightarrow \mathbf{F}\,\neg leader_i\right]$$

  Aus Gründen der Effizienz wird nicht verlangt, dass der aktive Leader in Anwesenheit einer inaktiven Entität zurücktritt, welche erst zu einem unbestimmten Zeitpunkt in der Zukunft aktiv wird. Deshalb die Prämisse, dass $j$ aktiv sein muss.

- „Ein neuer Leader ist stets besser als der vorherige“. Diese Eigenschaft stellt sicher, dass nachfolgende Leader stets eine höhere Identität haben als ihre Vorgänger:

$$\mathbf{G}\left[\forall i, j : (leader_i \wedge \neg \mathbf{X}\, leader_i \wedge \mathbf{X}\, \mathbf{F}\, leader_j) \rightarrow i \sqsubset j\right]$$

  Aufgrund dieser Eigenschaft ist auch sichergestellt, dass eine Entität, die einmal als Leader zurückgetreten ist, niemals wieder Leader werden kann.

## 4.4. LTL Model Checking

Wir wenden uns nun der Frage zu, wie man auf Basis der zuvor vorgestellten Temporallogik Eigenschaften von reaktiven Systemen verifizieren kann. Das hier vorgestellte Verfahren basiert im Wesentlichen auf den Ausführungen in [Meu06] und [Kat99].

### 4.4.1. Automatenmodelle

Bevor wir nun dazu übergehen können den LTL Model Checking Algorithmus vorzustellen, müssen zunächst die notwendigen Automatenmodelle vorgestellt werden. Das LTL Model Checking basiert dabei auf endlichen Automaten, die in der Lage sind unendliche Wörter[3] zu akzeptieren. Im Folgenden betrachten wir geeignete Automatenmodelle, beginnend mit einem Automatenmodell, welches in der Lage ist, endliche Wörter mit einer endlichen Menge von Zuständen zu akzeptieren.

**Definition 4.9 (Labeled Finite State Automata)** *Ein Labeled Finite State Automaton (kurz: LFSA) $A$ ist definiert als Sixtupel $(\Sigma, S, S_0, \rho, F, l)$. Hierbei ist*

- $\Sigma$ *das (nicht leere) Eingabealphabet,*
- $S$ *der endliche Zustandsraum,*
- $\emptyset \subset S_0 \subseteq S$ *die (nicht leere) Menge von Startzuständen,*
- $\rho : S \rightarrow \wp(S)$ *die Funktion, die jedem Zustand aus $S$ eine Menge von möglichen Folgezuständen zuordnet,*
- $F \subseteq S$ *die Menge der akzeptierenden Endzustände,*
- $l : S \rightarrow \Sigma$ *die Funktion, die jedem Zustand aus $S$ ein Zeichen aus $\Sigma$ zuordnet.*

[3] Hier wird zunächst nur allgemein von Wörtern als Zeichenfolgen über einem endlichen Alphabet gesprochen; der Zusammenhang zur LTL wird im nachfolgenden Abschnitt dargestellt.

Für jeden Zustand $s \in S$ ist $\rho(s)$ die Menge der Zustände, in die der Automat $A$ übergehen kann, wenn er sich im Zustand $s$ befindet. $\rho$ wird deshalb als die *Übergangsfunktion* bzw. *Übergangsrelation* (engl.: *transition relation*) des Automaten $A$ bezeichnet. Wir schreiben $s \to s'$ gdw. $s' \in \rho(s)$.

Die Funktion $l$ ordnet jedem Zustand $s \in S$ einen Markierung (engl. *label*) aus $\Sigma$ zu, und wird als *labeling function* bezeichnet. Ferner bezeichne $\Sigma^*$ die Menge der endlichen Wörter über $\Sigma$.

**Definition 4.10 (Deterministischer LFSA)** *Ein LFSA $A$ ist genau dann* deterministisch*, wenn für alle Markierungen $a \in \Sigma$ gilt*

$$|\{s \in S_0 \mid l(s) = a\}| \leq 1$$

*und für alle $a \in \Sigma$ und $s \in S$ gilt*

$$|\{s' \in \rho(s) \mid l(s') = a\}| \leq 1.$$

Also ist $A$ genau dann deterministisch, wenn die Anzahl gleichbezeichneter Startzustände höchstens eins ist, und für jede Markierung und jeden Zustand der Folgezustand eindeutig bestimmt ist.

Umgekehrt ist ein LFSA $A$ genau dann *nicht-deterministisch*, wenn mehreren Startzuständen die gleiche Markierung zugeordnet wird oder die Übergangsfunktion für einen Zustand mehrere mit der gleichen Markierung versehene Folgezustände liefert.

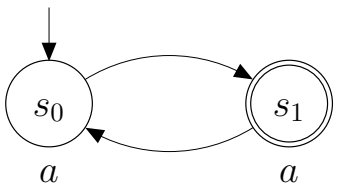
**Definition 4.11 (Lauf eines LFSA)** *Für einen LFSA $A$ ist ein* Lauf von $A$ *definiert als eine endliche Folge von Zuständen $\sigma = s_0 \dots s_n$ mit $s_0 \in S_0$ und $s_i \to s_{i+1}$ für $0 \leq i < n$. $\sigma$ heißt* akzeptierend *genau dann wenn $s_n \in F$.*

**Definition 4.12 (Akzeptierte Sprache eines LFSA)** *Ein LFSA $A$ akzeptiert ein endliches Wort $w = a_0 \dots a_n \in \Sigma^*$ genau dann wenn ein akzeptierender Lauf $\sigma = s_0 \dots s_n$ existiert mit $l(s_i) = a_i$ für $0 \leq i \leq n$. Die von $A$ akzeptierte Sprache $\mathcal{L}(A) \subseteq \Sigma^*$ ist*

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ akzeptiert } w\}.$$

Im speziellen Fall von $F = \emptyset$ existiert kein akzeptierender Lauf und es gilt $\mathcal{L}(A) = \emptyset$.

**Beispiel 4.3** *Sei $A = (\{a\}, \{s_0, s_1\}, \{s_0\}, \rho, \{s_1\}, l)$ mit $\rho(s_0) = \{s_1\}$, $\rho(s_1) = \{s_0\}$ und $l(s_0) = l(s_1) = a$ der nachfolgend dargestellte LFSA.*



*A akzeptiert offensichtlich die Menge der endlichen Wörter über* $\{a\}$*, deren Länge ein Vielfaches von* 2 *ist, also* $\mathcal{L}(A) = \{a^{2n} \mid n \in \mathbb{N}\}$.

**Definition 4.13 (Äquivalenz von Automaten)** *Zwei LFSAs* $A_1$ *und* $A_2$ *heißen äquivalent genau dann wenn* $\mathcal{L}(A_1) = \mathcal{L}(A_2)$.

Die Forderung, unendliche Wörter akzeptieren zu können, ist maßgeblich für das LTL Model Checking, da reaktive Programme und sequentielle Schaltungen üblicherweise nicht terminieren, und somit keine endliche, akzeptierende Folge von Zuständen existieren kann. Allerdings erfüllt das zuvor vorgestellte Automatenmodell des LFSA diese Anforderung nicht, da das Akzeptanzkriterium für Wörter über das Erreichen eines Endzustandes definiert ist.

### Labeled Büchi Automata

Eine Klasse von Automaten, welche diese Anforderung erfüllen, stellen die sogenannten *Labeled Büchi Automata (LBA)* dar. Ein Labeled Büchi Automaton ist eine Variante eines *Labeled Finite State Automaton (LFSA)* mit einem speziellen Akzeptanzkriterium für Wörter, dem sogenannten *Büchi Kriterium.*

**Definition 4.14 (Lauf eines LBA)** *Für einen LBA* $A$ *ist ein* Lauf von $A$ *definiert als eine unendliche Folge von Zuständen* $\sigma = s_0 s_1 \ldots$ *mit* $s_0 \in S_0$ *und* $s_i \to s_{i+1}$ *für* $i > 0$. *Sei* $\lim(\sigma)$ *die Menge der Zustände, die durch* $\sigma$ *unendlich oft besucht werden, so heißt* $\sigma$ akzeptierend *genau dann wenn* $\lim(\sigma) \cap F \neq \emptyset$.

Bezeichne $\Sigma^\omega$ die Menge der unendlichen Wörter über $\Sigma$. Ein LBA $A$ akzeptiert ein unendliches Wort $w = a_0 a_1 \ldots \in \Sigma^\omega$ genau dann wenn es einen akzeptierenden Lauf $\sigma = s_0 s_1 \ldots$ gibt mit $l(s_i) = a_i$ für $i > 0$. Die von dem Automaten $A$ akzeptierte unendliche Sprache[4] ist dann definiert durch:

$$\mathcal{L}_\omega(A) = \{w \in \Sigma^\omega \mid A \text{ akzeptiert } w\}$$

Im speziellen Fall von $F = \emptyset$ ist $\lim(\sigma) \cap \emptyset = \emptyset$ und es gilt $\mathcal{L}_\omega(A) = \emptyset$.

In Worten ausgedrückt besagt die obige Definition also, dass ein LBA $A$ alle unendlichen Wörter $w \in \Sigma^\omega$ akzeptiert, für die ein Lauf $\sigma$ existiert, welcher zumindests einen Endzustand aus $F$ unendlich oft besucht.

Die Äquivalenz von Büchi Automaten ist analog zur Äquivalenz von Automaten auf endlichen Wörtern im vorherigen Abschnitt definiert (vgl. Definition 4.13):

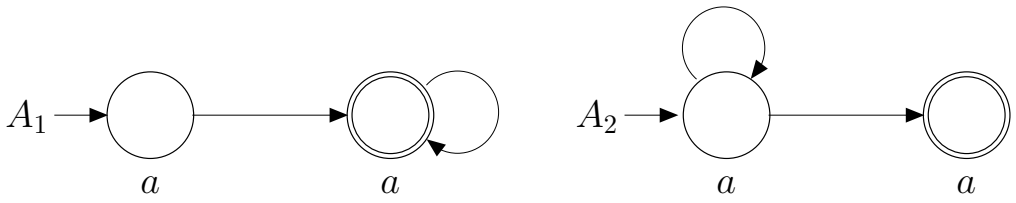
**Definition 4.15 (Äquivalenz von Büchi Automaten)** *Zwei LBAs* $A_1$ *und* $A_2$ *heißen äquivalent genau dann wenn* $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2)$.

[4]Der Terminus *unendlich* bezieht sich in diesem Fall nicht auf die Anzahl der Elemente der Sprache, sondern auf die Länge der Wörter.

Interessant ist hierbei insbesondere die Beziehung zwischen $\mathcal{L}(A)$, der Menge der endlichen Wörter, die von $A$ akzeptiert wird, und $\mathcal{L}_\omega(A)$, der Menge der unendlichen von $A$ akzeptierten Wörter. Denn zwei Automaten, die bezüglich $\mathcal{L}$ äquivalent sind, sind nicht zwangsläufig auch auf $\mathcal{L}_\omega$ äquivalent, wie im folgenden Beispiel zu sehen.

**Beispiel 4.4** *Seien $A_1$ und $A_2$ zwei Automaten, und bezeichne $\mathcal{L}(A_i)$ die Menge der endlichen und $\mathcal{L}_\omega(A_i)$ die Menge der unendlichen von $A_i$ akzeptierten Wörter für $i \in \{1, 2\}$.*

1. *Akzeptieren $A_1$ und $A_2$ die gleiche Menge endlicher Wörter, so kann die Menge der akzeptierten unendlichen Wörter dennoch unterschiedlich sein.*




   *Hier gilt $\mathcal{L}(A_1) = \mathcal{L}(A_2) = \{a^n \mid n \geq 2\}$, jedoch $\mathcal{L}_\omega(A_1) = \{a^\omega\} \neq \emptyset = \mathcal{L}_\omega(A_2)$.*

2. *Ebensowenig folgt aus der Äquivalenz bzgl. $\mathcal{L}_\omega$ die Äquivalenz bzgl. $\mathcal{L}$, was nicht unbedingt direkt ersichtlich ist. Aber basierend auf dem Automaten aus Beispiel 4.3 läßt sich ein einfaches Beispiel konstruieren.*


   *Es gilt*

$$\mathcal{L}_\omega(A_1) \;=\; \mathcal{L}_\omega(A_2) \;=\; \{a^\omega\}\,,$$

   *aber andererseits*

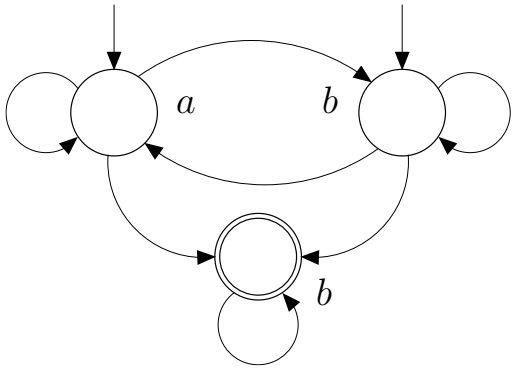
$$\mathcal{L}(A_1) \;=\; \left\{a^{2(n+1)} \mid n \in \mathbb{N}\right\} \;\neq\; \{a^{2n+1} \mid n \in \mathbb{N}\} \;=\; \mathcal{L}(A_2).$$

3. *Sind allerdings $A_1$ und $A_2$ deterministisch, dann gilt*

$$\mathcal{L}(A_1) = \mathcal{L}(A_2) \;\Rightarrow\; \mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2).$$

   *Die Umkehrung gilt, wie im vorherigen Beispiel gezeigt, nicht.*

In diesem Zusammenhang ist noch ein weiterer Unterschied zwischen LFSAs und LBAs interessant, der die Ausdrucksstärke betrifft. Während deterministische und nicht deterministische LFSAs die gleiche Ausdrucksstärke besitzen, sind nicht deterministische LBAs echt ausdrucksstärker als deterministische LBAs. Zum jedem deterministischen LFSA existiert ein deterministischer LFSA, welcher die gleiche Sprache akzeptiert (vgl. [Var95]). Der Algorithmus zur Transformation eines nicht deterministischen LFSAs in einen deterministischen wird als *Rabin-Scott subset construction* bezeichnet. Die Anzahl der Zustände im erzeugten deterministischen Automaten ist exponentiell in der Anzahl der Zustände des nicht deterministischen LFSAs. Für LBAs existiert kein solcher Algorithmus. D.h., es existiert ein nicht deterministischer LBA für den keine deterministische Version gefunden werden kann, welche die gleiche Sprache akzeptiert. Beispielsweise kann die Sprache $(a \mid b)^* b^\omega$ durch den folgenden nicht deterministischen LBA




akzeptiert werden, während kein deterministischer LBA existiert, welcher diese Sprache akzeptiert[5].

Wegen der unterschiedlichen Möglichkeiten die Akzeptanz von unendlichen Wörtern zu definieren, existieren in der Literatur auch diverse Varianten von Automatenmodellen über unendlichen Wörtern. Diese Automatenmodelle werden üblicherweise benannt nach den jeweiligen Forschern, die das Akzeptanzkriterium entworfen haben. Der Vollständigkeit wegen findet sich in Tabelle 4.1 eine Auflistung der bekanntesten Automatenmodelle für unendliche Wörter, wobei jeweils die Endzustandsmenge(n) und das Akzeptanzkriterum angegeben sind.

Für die Betrachtungen in diesem Dokument sind allerdings nur das *Büchi-* und das *Generalized Büchi-* Kriterium von Interesse. Letzterem ist der nachfolgende Abschnitt gewidmet.

### Generalized Labeled Büchi Automata

Eine Verallgemeinerung des Büchi Automatenmodells stellt der sogenannte *Generalized LBA* dar, der als Zwischenschritt des LTL Model Checking Algorithmus auftritt, und wie folgt definiert ist.

[5]Der Leser möge sich selbst davon überzeugen, dass diese Behauptung der Wahrheit entspricht.

| Name | Endzustandsmenge(n) | Akzeptanzkriterium |
|---|---|---|
| Büchi | $F \subseteq S$ | $\lim(\sigma) \cap F \neq \emptyset$ |
| Generalized Büchi | $\mathcal{F} = \{F_1, \ldots, F_k\}$ mit $F_i \subseteq S$ | $\forall i : \lim(\sigma) \cap F_i \neq \emptyset$ |
| Muller | $\mathcal{F} = \{F_1, \ldots, F_k\}$ mit $F_i \subseteq S$ | $\exists i : \lim(\sigma) \cap F_i \neq \emptyset$ |
| Rabin | $\mathcal{F} = \{(F_1, G_1), \ldots, (F_k, G_k)\}$ mit $F_i \subseteq S, G_i \subseteq S$ | $\exists i : \lim(\sigma) \cap F_i \neq \emptyset \wedge \lim(\sigma) \cap G_i = \emptyset$ |
| Street | $\mathcal{F} = \{(F_1, G_1), \ldots, (F_k, G_k)\}$ mit $F_i \subseteq S, G_i \subseteq S$ | $\forall i : \lim(\sigma) \cap F_i \neq \emptyset \wedge \lim(\sigma) \cap G_i = \emptyset$ |

Tabelle 4.1.: Automatenmodelle für unendliche Wörter

**Definition 4.16 (Generalized LBA)** *Ein Generalized Label Büchi Automaton (kurz: GLBA) $A$ ist definiert als Sixtupel $(\Sigma, S, S_0, \rho, \mathcal{F}, l)$, wobei $\Sigma$, $S$, $S_0$, $\rho$ und $l$ die gleiche Bedeutung wie im Falle des LBA zukommt, und $\mathcal{F}$ eine Menge von akzeptierenden Zustandsmengen $\{F_1, \ldots, F_k\}$ ist, mit $k \geq 0$ und $F_i \subseteq S$ für $i = 1, \ldots, k$.*

Statt einer einzigen Menge von akzeptierenden Endzuständen gibt es nun eine Menge von Mengen akzeptierender Endzustände $\mathcal{F} \subseteq \wp(S)$.

**Definition 4.17 (Lauf eines GLBA)** *Für einen GLBA $A$ heißt ein Lauf $\sigma = s_0 s_1 \ldots$* akzeptierend *genau dann wenn*

$$\forall i : [(0 < i \leq k) \Rightarrow \lim(\sigma) \cap F_i \neq \emptyset]$$

*gilt, also für jede Menge von akzeptierenden Endzuständen $F_i \in \mathcal{F}$ ein Zustand in $F_i$ existiert, der unendlich oft in $\sigma$ durchlaufen wird.*

Der Fall $\mathcal{F} = \emptyset$ bedeutet also hierbei, dass alle Läufe beliebig oft alle akzeptierenden Endzustände besuchen, folglich also jeder Lauf akzeptierend ist.

### Äquivalenz der Modelle

Man sieht leicht, dass zu jedem LBA ein äquivalenter GLBA angegeben werden kann, indem $\mathcal{F} = \{F\}$ gewählt wird. Wohl interessanter ist die Tatsache, dass man umgekehrt auch zu jedem GLBA einen äquivalenten LBA konstruieren kann. Die grundsätzliche Idee bei der Umwandlung eines GLBA $A$ mit $\mathcal{F} = \{F_1, \ldots, F_k\}$ in einen LBA $A'$ ist, $k$ Kopien von $A$ zu machen, eine für jede Menge $F_i \in \mathcal{F}$, und Zustände als Paare $(s, i)$ anzugeben mit $0 < i \leq k$.

**Definition 4.18 (Umwandlung eines GLBAs in einen LBA)** *Vorgegeben sei ein GLBA $A = (\Sigma, S, S_0, \rho, \mathcal{F}, l)$ mit $\mathcal{F} = \{F_1, \ldots, F_k\}$. Dann ist $A' = (\Sigma, S', S'_0, \rho', F', l')$ mit*

- $S' = S \times \{i \mid 0 < i \leq k\}$
- $S'_0 = S_0 \times \{i\}$ *für ein beliebiges* $i$ *mit* $0 < i \leq k$
- $(s,i) \rightarrow' (s',i)$ *gdw.* $s \rightarrow s'$ *und* $s \notin F_i$
  $(s,i) \rightarrow' (s',(i \textbf{ mod } k)+1)$ *gdw.* $s \rightarrow s'$ *und* $s \in F_i$
- $F' = F_i \times \{i\}$ *für ein beliebiges* $i$ *mit* $0 < i \leq k$
- $l'(s,i) = l(s)$

*ein zu* $A$ äquivalenter LBA.

Es gilt zu beachten, dass für die Definition der Anfangs- und Endzustände von $A'$ ein beliebiges $i$ gewählt werden kann, und somit der Automat $A'$ nicht eindeutig bestimmt ist.

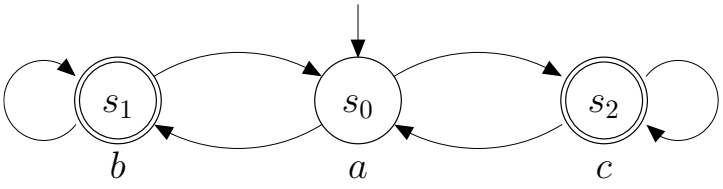
**Satz 4.1 (Äquivalenztheorem)** *Seien* $A$ *und* $A'$ *wie in Definition 4.18. Dann gilt:*

$$\mathcal{L}_\omega(A) \;=\; \mathcal{L}_\omega(A')$$

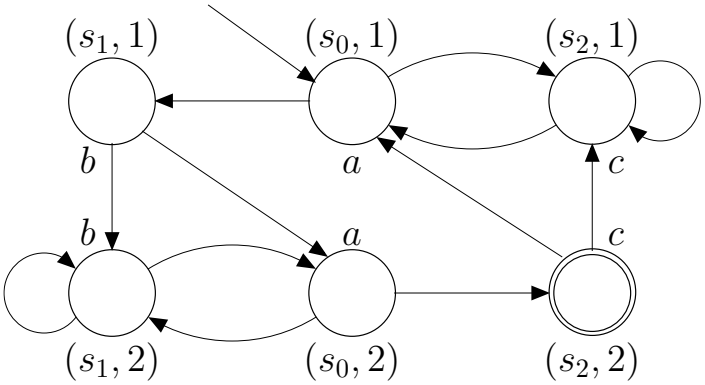
**Beweis:** Gemäß Definition 4.18 ist ein Lauf von $A'$ akzeptierend, wenn die Zustände $(s,i)$ mit $s \in F_i$ unendlich oft durchlaufen werden. Sobald ein Lauf einen solchen Zustand $(s,i)$ erreicht, geht der Automat $A'$ zur $(i+1)$-ten Kopie über. Von dieser Kopie kann die nächste Kopie über den Zustand $(s', i+1)$ mit $s' \in F_{i+1}$ erreicht werden, usw. $A'$ kann zu $(s,i)$ nur zurückkehren nachdem vorher alle $k$ Kopien durchlaufen wurden, und für jede Kopie ein akzeptierender Endzustand erreicht wurde. Damit ein Lauf einen Zustand $(s,i)$ unendlich oft durchlaufen kann, muss also aus jeder Kopie ein beliebiger akzeptierender Endzustand beliebig oft durchlaufen werden. Hiermit ist klar, dass $A$ und $A'$ die gleiche Sprache akzeptieren, folglich äquivalent sind. □

Betrachten wir nachfolgend ein Beispiel, um die Konstruktion des äquivalenten LBAs zu einem gegebenen GLBA zu verdeutlichen.

**Beispiel 4.5** *Sei* $A$ *der im folgenden dargestellte GLBA*




*mit zwei Mengen von akzeptierenden Endzuständen* $F_1 = \{s_1\}$ *und* $F_2 = \{s_2\}$*. Die Zustandsmenge des äquivalenten LBAs* $A'$ *ist* $S = \{s_0, s_1, s_2\} \times \{1,2\}$*. Eine mögliche Konstruktion für* $A'$ *ist*




*wobei* $(s_0, 1)$ *als Anfangszustand und* $(s_2, 2)$ *als akzeptierender Endzustand gewählt wurde. Jeder akzeptierende Lauf muss also* $(s_2, 2)$ *unendlich oft durchlaufen* ($s_2 \in F_2$). *Dazu muss allerdings auch ein mit* $s_1$ *markierter Zustand* ($s_1 \in F_1$) *unendlich oft durchlaufen werden.*

Die Anzahl der Zustände eines so konstruierten einfachen LBAs ist $\mathcal{O}(k \times |S|)$, wobei $S$ die Zustandsmenge des GLBA und $k$ die Anzahl der akzeptierenden Endzustandsmengen ist.

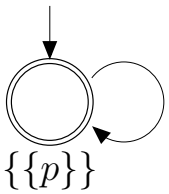
### 4.4.2. Automaten für LTL-Formeln

Basierend auf den vorgestellten Automatenmodellen für unendliche Wörter soll in diesem Abschnitt der Zusammenhang zwischen LTL-Formeln und Büchi Automaten dargestellt werden. Im Anschluss wird ein Algorithmus zur Konstruktion von Büchi Automaten für LTL-Formeln präsentiert.

Angenommen die Zustände eines LBA seien statt mit einzelnen Symbolen aus $\Sigma$ mit Mengen von Symbolen markiert, zum Beispiel $l : S \to \wp(\Sigma)$ für ein beliebiges Alphabet $\Sigma$. In diesem Fall wird ein Wort $w = a_0 a_1 \ldots$ genau dann akzeptiert, wenn ein akzeptierender Lauf $\sigma = s_0 s_1 \ldots$ existiert mit $a_i \in l(s_i)$ für alle $i \geq 0$.

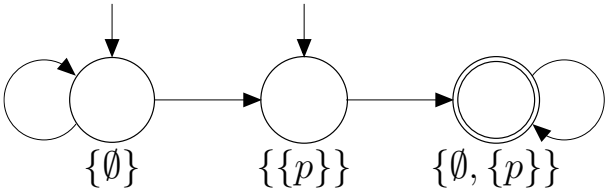
Sei nun $\Sigma = \wp(\mathcal{AP})$, wobei $\mathcal{AP}$ die Menge der Propositionalzeichen ist. Dann werden die Zustände mit Mengen von Mengen von Propositionalzeichen markiert, und Wörter bestehen aus einer Folge von Zeichen, wobei jedes Zeichen eine Menge von Propositionalzeichen ist. Ein LBA für eine LTL-Formel $\phi$ akzeptiert also alle unendlichen Wörter, d.h. Folgen von Mengen von Propositionalzeichen, die $\phi$ erfüllen. Dies soll in den folgenden Beispielen erläutert werden (vgl. [Kat99, S. 73f]).

**Beispiel 4.6** *Sei* $\mathcal{AP} = \{p\}$ *und A der im folgenden dargestellte LBA.*



*Jeder akzeptierende Lauf von A durchläuft den mit $\{\{p\}\}$ bezeichneten Zustand unendlich oft. Dementsprechend ist die von A akzeptierte unendliche Sprache $\mathcal{L}_\omega(A) = (\{\{p\}\})^\omega$, kurz $p^\omega$. Aus dieser Beobachtung folgt, dass die akzeptierenden Läufe von A exakt der Folge von Propositionalzeichen entsprechen, für die die LTL-Formel $\mathbf{G}p$ gilt.*

**Beispiel 4.7** *Sei nun wiederum $\mathcal{AP} = \{p\}$, und $A'$ der folgende LBA:*




*Dieser Automat kann entweder in einem Zustand starten, der p erfüllt (dem mit $\{\{p\}\}$ markierten Zustand), oder in einem Zustand, der p nicht erfüllt (dem mit $\{\emptyset\}$ markierten Zustand). Jeder akzeptierende Lauf muss den mit $\{\{p\}\}$ markierten Zustand einmalig durchlaufen und anschliessend unendlich oft den rechten, mit $\{\emptyset, \{p\}\}$ markierten Zustand, durchlaufen. Vereinfacht ausgedrückt akzeptiert $A'$ also jeden Lauf, für den mindestens einmal p gilt. Dies entspricht der LTL-Formel $\mathbf{F}p$.*

Aus diesen Beispielen ergibt sich, dass $\{\emptyset\}$ für die Formel $\neg p$, $\{\{p\}\}$ für die Formel $p$ und $\{\emptyset, \{p\}\}$ für die Formel $\neg p \vee p = \top$ steht. Allgemein stehen Mengen von Mengen von Propositionalzeichen also für aussagenlogische Formeln. Genauer gesagt, Mengen von Teilmengen einer gegebenen Menge $\mathcal{AP}$ von Propositionalzeichen dienen zur Kodierung von aussagenlogischen Formeln über $\mathcal{AP}$.

Formal ausgedrückt, seien $\mathcal{AP}_1, \ldots, \mathcal{AP}_n \subseteq \mathcal{AP}$, dann steht die Menge $\mathcal{AP}_i$ mit $0 < i \leq n$ für die Formel

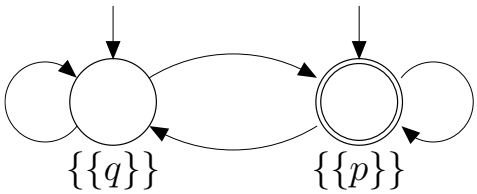
$$(\forall p \in \mathcal{AP}_i : p) \wedge (\forall p \in \mathcal{AP} \setminus \mathcal{AP}_i : \neg p),$$

geschrieben als $[\![\mathcal{AP}_i]\!]$. Die Menge $\{\mathcal{AP}_{k_1}, \ldots, \mathcal{AP}_{k_m}\}$ mit $m \geq 1$ und $0 < k_j \leq n$ kodiert nun die aussagenlogische Formel

$$[\![\mathcal{AP}_{k_1}]\!] \vee \ldots \vee [\![\mathcal{AP}_{k_m}]\!].$$

Man beachte, dass die Mengen von Mengen von Propositionalzeichen nicht leer sein dürfen.

**Beispiel 4.8** *Betrachten wir nun den folgenden Automaten mit $\mathcal{AP} = \{p, q\}$.*



*Basierend auf den vorhergehenden Erkenntnissen können wir die Markierung des linken Zustands als $\neg p \wedge q$ und die Markierung des rechten Zustands als $p \wedge \neg q$ interpretieren. Jeder akzeptierende Lauf des Automaten muss folglich den Zustand, der $p \wedge \neg q$ erfüllt, unendlich oft durchlaufen, wobei jeweils zwischen zwei aufeinanderfolgenden Besuchen des zweiten Zustands, der erste Zustand, welcher $\neg p \wedge q$ erfüllt, beliebig oft durchlaufen werden kann. Folglich entsprechen die akzeptierenden Läufe des Automaten exakt den Folgen von Mengen von Propositionalzeichen, die die LTL-Formel*

$$\mathbf{G}\,[(\neg p \wedge q)\ \mathbf{U}\ (p \wedge \neg q)]$$

*erfüllen.*

Es liegt nun die Vermutung nahe, dass sich zu jeder LTL-Formel (über einer gegebenen Menge von Propositionalzeichen $\mathcal{AP}$) ein Büchi Automat konstruieren lässt, und tatsächlich gilt der folgende Satz:

**Satz 4.2** *Zu jeder LTL-Formel $\phi$ kann ein Büchi Automat $A$ über dem Alphabet $\Sigma = \wp(\mathcal{AP})$ konstruiert werden, so dass die akzeptierte Sprache $\mathcal{L}_\omega(A)$ den Folgen von Mengen von Propositionalzeichen entspricht, die $\phi$ erfüllen.*

**Beweis:** Der Beweis dieses Theorems findet sich in [WVS83]. □

Basierend auf diesem Ergebnis lässt sich dann ein Verfahren zum LTL Model Checking formulieren. Der einfachste, aber auch naivste Ansatz ist in Tabelle 4.2 dargestellt.

Hierbei werden zunächst Büchi Automaten für die Spezifikation – notiert als LTL-Formel – und für das Modell des Systems erstellt, und anschließend geprüft, ob sämtliche Verhaltenweisen von $A_{sys}$ erwünscht sind, also durch die Spezifikation $A_\phi$ abgedeckt werden. Dieser Ansatz erscheint zunächst einmal einfach genug für eine Realisierung. Allerdings ist der Sprachvergleich der beiden Automaten $A_{sys}$ und $A_\phi$, also die Berechnung von $\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi)$, PSPACE-vollständig und es handelt sich folglich um ein schwieriges Problem aus der Klasse NP[6].

Glücklicherweise lässt sich der Test auf Mengeninklusion mittels der Umformung

$$\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi) \quad \Leftrightarrow \quad \mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(\bar{A_\phi}) = \emptyset$$

vermeiden, so dass stattdessen lediglich die Schnittmenge auf Leerheit geprüft werden muss. Hierbei steht $\bar{A}_\phi$ für das Komplement von $A_\phi$, also den Automaten, der die Sprache

[6]PSPACE-vollständige Probleme gehören der Kategorie von Problemen an, die noch mit polynomiellem Speicherplatzbedarf gelöst werden können, d.h. Speicherplatzbedarf, welcher polynomiell in der Größe des Problems ist (zum Beispiel der Anzahl der Zustände eines Büchi Automaten). Für diese Probleme ist es sehr unwahrscheinlich einen Algorithmus zu finden, welcher nicht exponentielle Laufzeitkomplexität hat.

1. Konstruktion des Büchi Automaten für $\phi$, $A_\phi$
2. Konstruktion des Büchi Automaten für das Modell des Systems, $A_{sys}$
3. Prüfung ob $\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi)$

Tabelle 4.2.: Naiver LTL Model Checking Algorithmus

$\Sigma^\omega \backslash \mathcal{L}_\omega(A)$ akzeptiert. Allerdings ist $\bar{A_\phi}$ nach der Konstruktion sehr groß[7]: Hat zum Beispiel $A_\phi$ $n$ Zustände, so hat $\bar{A_\phi}$ $c^{n^2}$ Zustände, für ein $c > 1$.

Das Komplement des Automaten $A_\phi$ ist aber gerade äquivalent zu dem Automaten $A_{\neg\phi}$, das heisst es gilt $\mathcal{L}_\omega(\bar{A_\phi}) = \mathcal{L}_\omega(A_{\neg\phi})$. Diese Erkenntnis ermöglicht es einen effizienten Algorithmus für das LTL Model Checking anzugeben, wie in Tabelle 4.3 gezeigt.

1. Konstruktion des Büchi Automaten für $\neg\phi$, $A_{\neg\phi}$
2. Konstruktion des Büchi Automaten für das Modell des Systems, $A_{sys}$
3. Prüfung ob $\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\phi}) = \emptyset$

Tabelle 4.3.: Effizienter LTL Model Checking Algorithmus

Die zugrundeliegende Idee hierbei entspricht der Überlegung, einen Automaten $A_{\neg\phi}$ zu konstruieren, der genau das *unerwünschte* Verhalten repräsentiert. Anschließend wird geprüft, ob $A_{sys}$ keine dieser unerwünschten Verhaltensweisen beinhaltet, also ob die Schnittmenge der beiden Automaten leer ist. Ist dies der Fall, entspricht das Modell des Systems der Spezifikation.

Oder anders ausgedrückt, hat $A_{sys}$ einen akzeptierenden Lauf, der zugleich auch ein akzeptierender Lauf von $A_{\neg\phi}$ ist, so ist dies ein Beispiel für einen Lauf, der $\phi$ widerspricht. Existiert kein solcher Lauf so ist die Spezifikation $\phi$ erfüllt. In [EL85] findet sich ein Beweis, dass dieser dritte Schritt, also der Test $\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\phi}) = \emptyset$, damit in linearer Zeit entscheidbar ist, und somit nicht mehr in die Klasse NP fällt.

[7] Für deterministische Büchi Automaten existiert ein Algorithmus, der polynomiell in der Größe des Automaten ist, allerdings sind deterministische Büchi Automaten auch weitaus weniger mächtig als nicht-deterministische.

### 4.4.3. Von LTL-Formeln zu Büchi Automaten

Das Ziel ist es einen Büchi Automaten zu konstruieren, der in der Lage ist alle unendlichen Zustandsfolgen zu generieren, die eine gegebene LTL-Formel $\phi$ erfüllen. Der hier beschriebene Algorithmus ist in Abbildung 4.1 schematisch dargestellt und entspricht dem in [GPVW95] und [Kat99] angegebenen Algorithmus.

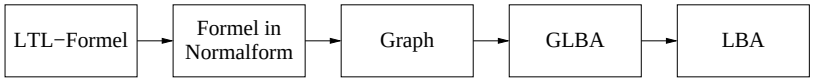



Abbildung 4.1.: Übersicht über den LTL-zu-LBA-Algorithmus

#### Normalform von LTL-Formeln

Der erste Schritt des Algorithmus besteht darin, die gegebene LTL-Formel in eine sogenannte *Normalform* zu bringen. Dazu ist zunächst anzumerken, dass die in Definition 4.1 vorgestellte Syntax von LTL zwar adäquat, aber für die weiteren Schritte des LTL Model Checking Algorithmus nicht direkt geeignet ist. Die Idee ist nun, eine Teilmenge von LTL-Formeln zu definieren, die einerseits adäquat ist und andererseits nur diejenigen Formeln enthält, die sich auf einfache Weise in einem Automaten umsetzen lassen.

**Definition 4.19 (Normalform von LTL-Formeln)** *Sei $\mathcal{AP}$ eine Menge von Propositionalzeichen. Dann ist die Menge $\mathcal{L}_{LTL}^{Norm}$ der gültigen LTL-Formeln in* Normalform *induktiv definiert durch:*

1. *$p$ und $\neg p$ sind in Normalform ($p \in \mathcal{AP}$).*
2. *Ist $\phi$ in Normalform, so ist auch $\mathbf{X}\,\phi$ in Normalform.*
3. *Sind $\phi, \psi$ in Normalform, so auch $\phi \vee \psi$, $\phi \wedge \psi$, $\phi\,\mathbf{U}\,\psi$ und $\phi\,\mathbf{R}\,\psi$.*

Eine Formel $\phi$ ist also in Normalform, wenn Negationen nur direkt vor einem Propositionalzeichen stehen und $\phi$ keinen der temporalen Operatoren **F** und **G** enthält. Letzteres kann erreicht werden, indem alle Vorkommen von $\rightarrow$, **F** und **G** wie folgt elimiert werden.

$$\begin{aligned} \phi \rightarrow \psi &\equiv \neg\phi \vee \psi \\ \mathbf{F}\,\phi &\equiv \top\,\mathbf{U}\,\phi \\ \mathbf{G}\,\phi &\equiv \bot\,\mathbf{R}\,\phi \end{aligned}$$

Desweiteren müssen alle Vorkommen von $\top$ und $\bot$ eliminiert werden. Für ein beliebiges $p \in \mathcal{AP}$ kann $\top$ durch $p \vee \neg p$ und $\bot$ durch $p \wedge \neg p$ ersetzt werden.

Nachdem alle Vorkommen von $\rightarrow$, $\mathbf{F}$, $\mathbf{G}$, $\top$ und $\bot$ in einer LTL-Formel wie oben angegeben ersetzt wurden, kann diese mit Hilfe der folgenden Äquivalenzen in Normalform überführt werden:

$$\begin{array}{rcl} \neg(\phi \vee \psi) & \equiv & (\neg\phi) \wedge (\neg\psi) \\ \neg(\phi \wedge \psi) & \equiv & (\neg\phi) \vee (\neg\psi) \\ \neg\mathbf{X}\,\phi & \equiv & \mathbf{X}\,(\neg\phi) \\ \neg(\phi\,\mathbf{U}\,\psi) & \equiv & (\neg\phi)\,\mathbf{R}\,(\neg\psi) \\ \neg(\phi\,\mathbf{R}\,\psi) & \equiv & (\neg\phi)\,\mathbf{U}\,(\neg\psi) \end{array}$$

Liest man die obigen Äquivalenzen von links nach rechts, so stellt man fest, dass jeweils die *äußerste* Negation nach *innen* verschoben wird. Es ist offensichtlich, dass durch iterative Anwendung der Äquivalenzumformungen Negationen soweit wie möglich nach *innen* verschoben werden, d.h. am Ende stehen Negationen nur noch vor Propositionalzeichen.

**Lemma 4.1** *$\mathcal{L}_{LTL}^{Norm}$ ist eine adäquate Menge von LTL Formeln.*

**Beweis:** Bleibt als Übung für den Leser. □

**Beispiel 4.9** *Als Beispiel betrachten wir hier die Überführung der LTL-Formel $\phi = \mathbf{G}(p \wedge \neg\mathbf{X}q)$ mit $\mathcal{AP} = \{p, q\}$ in Normalform.*

$$\begin{array}{rl} & \mathbf{G}(p \wedge \neg\mathbf{X}q) \\ \equiv & \neg\mathbf{F}\neg(p \wedge \neg\mathbf{X}q) \\ \equiv & \neg(\top\,\mathbf{U}\,\neg(p \wedge \neg\mathbf{X}q)) \\ \equiv & \bot\mathbf{R}(p \wedge \neg\mathbf{X}q) \\ \equiv & (p \wedge \neg p)\mathbf{R}(p \wedge \neg\mathbf{X}q) \\ \equiv & (p \wedge \neg p)\mathbf{R}(p \wedge \mathbf{X}\neg q) \end{array}$$

*Die Formel $(p \wedge \neg q)\mathbf{R}(p \wedge \mathbf{X}\neg q)$ erfüllt die Normalform-Bedingungen, da die Negation nur noch vor $p$ und $q$ steht und als temporale Operatoren nur noch $\mathbf{X}$ und $\mathbf{R}$ vorkommen.*

Die Zeit, welche für die Überführung einer LTL-Formel $\phi$ in Normalform benötigt wird, ist linear in der Größe von $\phi$, wobei die Größe einer LTL-Formel wie folgt definiert ist.

**Definition 4.20 (Größe von LTL-Formeln)** *Sei $p \in \mathcal{AP}$ und $\phi, \psi$ gültige LTL-Formeln.*

*Die Funktion* $|\cdot| : \mathcal{L}_{LTL} \to \mathbb{N}$*, induktiv definiert durch*

$$\begin{aligned}
|p| &= 1 \\
|\neg\phi| &= 1 + |\phi| \\
|\phi \vee \psi| &= 1 + |\phi| + |\psi| \\
|\phi \wedge \psi| &= 1 + |\phi| + |\psi| \\
|\phi \rightarrow \psi| &= 1 + |\phi| + |\psi| \\
|\mathbf{X}\,\phi| &= 1 + |\phi| \\
|\phi\,\mathbf{U}\,\psi| &= 1 + |\phi| + |\psi| \\
|\phi\,\mathbf{R}\,\psi| &= 1 + |\phi| + |\psi| \\
|\mathbf{F}\,\phi| &= 1 + |\phi| \\
|\mathbf{G}\,\phi| &= 1 + |\phi|,
\end{aligned}$$

*ordnet jeder LTL-Formel* $\phi \in \mathcal{L}_{LTL}$ *eine Größe zu.*

Der Algorithmus für die Überführung einer LTL-Formel $\phi$ in die entsprechende Normalform hat eine *worst-case Laufzeit* von $\mathcal{O}(|\phi|)$. Bei der Beschreibung der nachfolgenden Schritte des Algorithmus wird davon ausgegangen, dass LTL-Formeln bereits in Normalform vorliegen.

### Graphkonstruktion

Der nächste Schritt des Algorithmus besteht in der Konstruktion eines Graphen für eine gegebene LTL-Formel $\phi$. Der Quelltext der Funktion *CreateGraph* wird, um unabhängig von einer speziellen Programmiersprache zu bleiben, in einer abstrakten Notation angegeben[8], die auf Dijkstra's Guarded Command Language [Dij75] basiert.

Ein Graph $\mathcal{G}$ ist ein Paar $(V, E)$ mit einer Menge von Knoten $V$ und einer Menge von Kanten $E \subseteq V \times V$. In unserem Anwendungsfall seien die Knoten des Graphen wie folgt definiert:

**Definition 4.21 (Knoten)** *Ein Knoten* $v$ *ist ein Quadrupel* $(P, N, O, Sc)$*, wobei*

- $P \subseteq V \cup \{init\}$ *die Menge der Vorgänger ist, und*
- $N, O, Sc$ *sind Mengen von LTL-Formeln (in Normalform).*

Für $v = (P, N, O, Sc)$ schreiben wir $P(v) = P$, $N(v) = N$, $O(v) = O$ und $Sc(v) = Sc$. $P$ (Predecessors) ist die Menge der Vorgänger eines Knotens, und definiert damit die Struktur des Graphen. $N(v) \cup O(v)$ sind die für $v$ zu prüfenden Formeln, wobei $N$ (New) die Menge der Formeln repräsentiert, die noch nicht bearbeitet wurden, und $O$ (Old) die

[8] Anhang A enthält eine detailierte Beschreibung der Notation.

bereits bearbeiteten. $Sc$ (Successors) beinhaltet diejenigen Formeln, die für alle Knoten gelten müssen, die direkte Nachfolger von Knoten sind, welche die Formeln in $O$ erfüllen.

*init* ist ein spezieller Bezeichner, der keinem realen Knoten in $\mathcal{G}$ entspricht, d.h. $init \notin V$, sondern als Markierung für Startknoten im Graphen dient. Ein Knoten $v$ wird als Startknoten bezeichnet gdw. $init \in P(v)$.

Der in Listing 4.1 abgedruckte Algorithmus zur Graphkonstruktion und die folgenden Erläuterungen zum Algorithmus sind im wesentlichen identisch zu dem in [Kat99] enthaltenen Algorithmus, mit Ausnahme kleinerer Korrekturen und stilistischer Anpassungen.

Der Algorithmus konstruiert den Graphen $\mathcal{G}_\phi$ in einem Tiefendurchlauf, beginnend bei dem mit der Eingabeformel $\phi$ markierten Knoten. Die Liste $S$ enthält alle Knoten, die bisher noch nicht vollständig erforscht worden sind. Zu Beginn enthält $S$ nur den mit $\phi$ markierten Knoten *init*. Die Menge $Z$ enthält die bereits erzeugten Knoten für den Graphen. Der Algorithmus terminiert sobald alle Knoten vollständig erforscht sind ($S = \langle\rangle$), und $Z$ enthält anschließend den fertig konstruierten Grahpen. Ein Knoten $v$ gilt als vollständig erforscht wenn alle Formeln, die in $v$ gelten müssen, geprüft worden sind ($N(v) = \emptyset$).

Sei $N(v) = \emptyset$ für einen Knoten $v$. Dann wird zunächst geprüft, ob $Z$ bereits einen Knoten mit identischem $O$ und $Sc$ enthält. Falls ja wird kein neuer Knoten erstellt. Stattdessen wird vermerkt, dass der existierende Knoten nun zusätzlich über alle Vorgänger von $v$ erreichbar ist. Existiert noch kein solcher Knoten, wird $v$ zu $Z$ hinzugefügt und die Nachfolger von $v$ müssen erforscht werden. Hierzu wird ein neuer mit $Sc(v)$ markierter Knoten zu $S$ hinzugefügt. Anschließend wird $v$ aus $S$ entfernt, da $v$ ja bereits vollständig erforscht ist.

Anderenfalls, wenn $N(v) \neq \emptyset$, gibt es noch Formeln für $v$, die geprüft werden müssen. In jedem Schritt wird nun eine Formel $\psi$ aus $N(v)$ entfernt, und nach folgenden Fällen unterschieden bearbeitet.

- Wenn $\psi$ die Negation eines Propositionalzeichens ist, welches bereits zuvor bearbeitet worden ist, haben wir einen Widerspruch gefunden, und der Knoten $v$ wird nicht weiter erforscht ($S := S'$).

- Falls $\psi$ ein Propositionalzeichen ist, dessen Negation noch nicht bearbeitet worden ist, ist nichts zu tun.

- Ist $\psi = \psi_1 \wedge \psi_2$, dann müssen, damit $\psi$ erfüllt wird, sowohl $\psi_1$ als auch $\psi_2$ erfüllt sein. D.h. $\psi_1$ und $\psi_2$ werden, sofern sie nicht bereits geprüft worden sind, also $\psi_1 \notin O(v)$ bzw. $\psi_2 \notin O(v)$, zu $N(v)$ hinzugenommen.

- Für $\psi = \mathbf{X}\varphi$ muss für alle direkten Nachfolger $\varphi$ erfüllt sein, folglich wird $\varphi$ zu $Sc(v)$ hinzugefügt.

- Bleiben noch die Fälle Disjunktion, **U**-Formel und **R**-Formel zu betrachten. Hierbei wird der Knoten $v$ jeweils aufgespalten in zwei Knoten $w_1$ und $w_2$. Die beiden

```
function CreateGraph (φ: Formula): set of Vertex;
(* Vorbedingung: φ ist LTL–Formel in Normalform *)
begin var S: sequence of Vertex,
          Z: set of Vertex, (* bereits erforschte Knoten *)
          w1, w2: Vertex,
          ψ: Formula;
  S, Z := ⟨({init}, {φ}, ∅, ∅)⟩, ∅;
  do S = ⟨v⟩⌢S' ⟶
    (* v ist erster Knoten, S' ist Rest *)
    if N(v) = ∅ ⟶ (* alle Formeln von v erforscht *)
      if (∃w ∈ Z : Sc(v) = Sc(w) ∧ O(v) = O(w)) ⟶
        P(w), S := P(w) ∪ P(v), S' (* w ist Kopie von v *)
      [] ¬(∃w ∈ Z : ...) ⟶
        S, Z := ⟨({v}, Sc(v), ∅, ∅)⟩⌢S', Z ∪ {v}
      fi
    [] N(v) ≠ ∅ ⟶ (* noch Formeln von v zu erforschen *)
      let ψ in N(v);
      N(v) := N(v) \ {ψ};
      if ψ ∈ AP ∨ (¬ψ) ∈ AP ⟶
        if (¬ψ) ∈ O(v) ⟶ S := S'
        [] (¬ψ) ∉ O(v) ⟶ skip
        fi
      [] ψ = (ψ1 ∧ ψ2) ⟶ N(v) := N(v) ∪ ({ψ1, ψ2} \ O(v))
      [] ψ = X φ ⟶ Sc(v) := Sc(v) ∪ {φ}
      [] ψ ∈ {ψ1 U ψ2, ψ1 R ψ2, ψ1 ∨ ψ2} ⟶
        (* ψ aufspalten *)
        w1, w2 := v, v;
        N(w1) := N(w1) ∪ (F1(ψ) \ O(w1));
        N(w2) := N(w2) ∪ (F2(ψ) \ O(w2));
        O(w1), O(w2) := O(w1) ∪ {ψ}, O(w2) ∪ {ψ};
        S := ⟨w1⟩⌢(⟨w2⟩⌢S')
      fi;
      O(v) := O(v) ∪ {ψ}
    fi
  od;
  return Z;
end
(* Nachbedingung: Z ist Menge der Knoten des Graphen *)
(* G_φ wobei die Startknoten durch init ∈ P und die Kanten *)
(* durch die P–Komponenten der Knoten bestimmt sind *)
```

Listing 4.1: Algorithmus zur Konstruktion eines Graphen für $\phi$

Knoten entsprechen den beiden Möglichkeiten die Formel $\psi$ zu erfüllen, jeweils in Abhängigkeit von der Struktur von $\psi$. Grundsätzlich wird $\psi$ dazu in eine Disjunktion umgeformt, so dass beide Teilformeln einzeln geprüft werden können.

- $\psi = \psi_1 \vee \psi_2$. Um $\psi$ zu erfüllen genügt es, dass entweder $\psi_1$ erfüllt wird oder $\psi_2$. Mit $F_i(\psi) = \{\psi_i\}$ für $i = 1, 2$ reduziert das die Prüfung auf $\psi_1$ (in $w_1$) oder $\psi_2$ (in $w_2$).
- $\psi = \psi_1 \,\mathbf{U}\, \psi_2$ läßt sich umschreiben in disjunktiver Form (vgl. [HR04, S. 184ff]) als $\psi_2 \vee [\psi_1 \wedge \mathbf{X}\ (\psi_1 \,\mathbf{U}\, \psi_2)]$.
- $\psi = \psi_1 \,\mathbf{R}\, \psi_2$ läßt sich ebenfalls umschreiben in disjunktiver Form als $(\psi_2 \wedge \psi_1) \vee [\psi_2 \wedge \mathbf{X}\,(\psi_1 \,\mathbf{R}\, \psi_2)]$.

**Beispiel 4.10** *Abbildung 4.2 zeigt das Ergebnis der Anwendung von CreateGraph auf die LTL-Formel $p$ $\mathbf{U}$ $q$ mit $p, q \in AP$.*

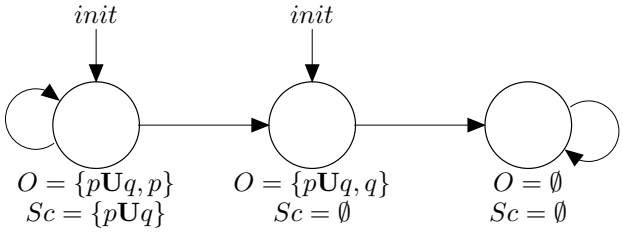



Abbildung 4.2.: Ergebnis von $CreateGraph(p\ \mathbf{U}\ q)$

Abschließend folgt wie üblich eine kurze Analyse der Laufzeitkomplexität der *CreateGraph*-Funktion: Prinzipiell kann jede Menge von Teilformeln von $\phi$ zu einem Knoten im Graphen $\mathcal{G}_\phi$ werden, so dass die Anzahl der Knoten im *worst-case* proportional zur Anzahl der Teilmengen von Formeln von $\phi$ ist. Die Anzahl der Teilformeln einer Formel ist aber gerade exponentiell in der Länge der Formel, so dass die Anzahl der Knoten schlimmstenfalls proportional zu $2^{|\phi|}$ ist. Es ergibt sich folglich eine Laufzeitkomplexität von $\mathcal{O}(2^{|\phi|})$ für die Konstruktion des Graphen.

### Konstruktion des GLBA

Der dritte Schritt des Model Checking Algorithmus ist die Konstruktion eines GLBA für den zuvor konstruierten Graphen. Die folgende Definition fasst beide Schritte zusammen.

**Definition 4.22 (Konstruktion eines GLBA für eine LTL-Formel)** *Sei $\phi$ eine LTL-Formel in Normalform. Dann ist der zugehörige GLBA $A = (\Sigma, S, S_0, \rho, \mathcal{F}, l)$ definiert durch*

- $\Sigma = \wp(\mathcal{AP})$

- $S = CreateGraph(\phi)$
- $S_0 = \{s \in S \mid init \in P(s)\}$
- $s \rightarrow s' \Leftrightarrow s \in P(s') \wedge s \neq init$
- $\mathcal{F} = \{\{s \in S \mid \phi_1 \mathbf{U} \phi_2 \notin O(s) \ \vee \ \phi_2 \in O(s)\} \mid \phi_1 \mathbf{U} \phi_2 \in Sub(\phi)\}$
- $l(s) = \{\mathcal{P} \subseteq AP \mid Pos(s) \subseteq \mathcal{P} \ \wedge \ \mathcal{P} \cap Neg(s) = \emptyset\}$,

*wobei $Sub(\phi)$ die Menge aller Teilformeln von $\phi$ darstellt, $Pos(s) = O(s) \cap AP$ die Menge der Propositionalzeichen, die erfüllt sind in $s$, und $Neg(s) = \{p \in AP \mid \neg p \in O(s)\}$ die Menge der negierten Propositionalzeichen, die erfüllt sind in $s$.*

Für die Zustandsmenge des Automaten $A_\phi$ wird die Menge der Knoten des durch *CreateGraph* erzeugten Graphen $\mathcal{G}_\phi$ gewählt. Die Startzustände entsprechen den Startknoten, also den Knoten $v$ für die $init \in P(v)$ gilt. Ein Übergang von einem Zustand $s$ in einen Zustand $s'$ ist genau dann möglich, wenn $s$ ein Vorgänger von $s'$ ist und $s$ nicht der spezielle Knoten *init* ist. Für jede Teilformel der Form $\phi_1 \mathbf{U} \phi_2$ in $\phi$ wird eine Menge von akzeptierenden Endzuständen erzeugt, die alle Zustände $s$ enthält, für die entweder $\phi_2 \in O(s)$ oder $\phi_1 \mathbf{U} \phi_2 \notin O(s)$ gilt. Die Anzahl der akzeptierenden Zustandsmengen von $A_\phi$ ist daher gleich der Anzahl der **U**-Teilformeln in $\phi$. Ein Zustand $s$ ist markiert mit allen Mengen von Propositionalzeichen, die die gültigen Aussagen in $s$ enthalten und keine der nicht gültigen Aussagen.

**Satz 4.3** *Ein gemäß Listing 4.1 und Definition 4.22 für eine LTL-Formel $\phi$ in Normalform konstruierter GLBA $A_\phi$ akzeptiert genau diejenigen Folgen über $\left(\wp(AP)\right)^\omega$, die die Formel $\phi$ erfüllen.*

**Beweis:** Der Beweis dieses Satzes findet sich in [GPVW95]. □

In [GPVW95] findet sich darüber hinaus eine Aufstellung über die Größenordnung einiger mit dem obigen Verfahren konstruierter Automaten für bestimmte LTL-Formeln, die exemplarisch in Tabelle 4.4 wiedergegeben ist.

Interessant zu beobachten ist hierbei, dass für die Formel $\neg(\phi_1 \,\mathbf{U}\, (\phi_2 \,\mathbf{U}\, \phi_3))$ keine Endzustandsmengen erzeugt werden, was darauf zurückzuführen ist, dass die Normalform-Darstellung keine **U**-Formeln mehr enthält[9]. Folglich ist jeder Lauf dieses Automaten akzeptierend.

[9] Dies zu überprüfen bleibt als Übung für den Leser.

| LTL-Formel | Zustände ($S$) | Übergänge ($\rho$) | Endzustandsm. ($\mathcal{F}$) |
|---|---|---|---|
| $\phi_1 \,\mathbf{U}\, \phi_2$ | 3 | 4 | 1 |
| $\phi_1 \,\mathbf{U}\, (\phi_2 \,\mathbf{U}\, \phi_3)$ | 4 | 6 | 2 |
| $\neg(\phi_1 \,\mathbf{U}\, (\phi_2 \,\mathbf{U}\, \phi_3))$ | 7 | 15 | 0 |
| $\mathbf{G}\,\mathbf{F}\,\phi_1 \rightarrow \mathbf{G}\,\mathbf{F}\,\phi_2$ | 9 | 15 | 2 |
| $\mathbf{F}\,\phi_1 \,\mathbf{U}\, \mathbf{G}\,\phi_2$ | 8 | 15 | 2 |
| $\mathbf{G}\,\phi_1 \,\mathbf{U}\, \phi_2$ | 5 | 6 | 1 |
| $\neg(\mathbf{F}\,\mathbf{F}\,\phi_1 \leftrightarrow \mathbf{F}\,\phi_2)$ | 22 | 41 | 2 |

Tabelle 4.4.: Größenordnungen von GLBAs für LTL-Formeln

### 4.4.4. Checking for emptiness

Wir haben gesehen, wie zu einer LTL-Formel $\phi$ ein Büchi-Automat $A_\phi$ konstruiert werden kann. Im Folgenden nehmen wir an, dass auch zu dem Modell des Systems, welches verifiziert werden soll, ein Büchi-Automat $A_{sys}$ existiert. Dann ist gemäß dem in Tabelle 4.3 dargestellten Algorithmus der letzte Schritt nun zu prüfen, ob die von den Automaten akzeptierten Läufe disjunkt sind, d.h. ob

$$\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\phi}) \quad = \quad \emptyset$$

gilt. $\mathcal{L}_\omega(A_{\neg\phi})$ repräsentiert alle Berechnungen, welche $\neg\phi$ erfüllen, d.h. es charakterisiert genau diejenigen Berechnungen, welche die Spezifikation $\phi$ verletzen. Anders ausgedrückt, $A_{\neg\phi}$ beschreibt das *unerwünschte Verhalten*. $\mathcal{L}_\omega(A_{sys})$ beschreibt alle möglichen Verhaltensweisen des Systems. Das Modell *sys* erfüllt demzufolge die Spezifikation $\phi$, wenn keine Berechnung existiert, die $\neg\phi$ erfüllen würde, also wenn kein unerwünschtes Verhalten eintritt.

1. Konstruktion des Produktautomaten $A_{sys} \otimes A_{\neg\phi}$, welcher die Sprache $\mathcal{L}_\omega(A_{sys} \cap \mathcal{L}_\omega(A_{\neg\phi})$ akzeptiert, und
2. prüfen ob $\mathcal{L}_\omega(A_{sys} \otimes A_{\neg\phi}) = \emptyset$.

Tabelle 4.5.: Verfahren zur Lösung des Leerheitsproblems

Der Test, ob $A_{sys}$ und $A_{\neg\phi}$ gemeinsame akzeptierende Läufe haben, besteht aus den in Tabelle 4.5 aufgeführten Schritten. Zunächst wird ein neuer Automat konstruiert, welcher die Schnittmenge der Sprachen von $A_{sys}$ und $A_{\neg\phi}$ akzeptiert. Diesen Automat bezeichnet man als den *synchronen Produkautomaten* (engl.: *synchronous product automaton*). Da Büchi Automaten unter Produktbildung abgeschlossen sind, ist garantiert,

dass ein Produktautomat stets existiert. Das Problem zu prüfen, ob ein gegebener Automat die leere Sprache akzeptiert ist als *Leerheitsproblem* (engl.: *emptiness problem*) bekannt.

### Produktautomaten auf endlichen Wörtern

Um ein besseres Verständnis der Konstruktion von Produktautomaten für LBAs zu vermitteln, wird hier einleitend wieder zunächst auf die Automaten für endliche Wörter eingegangen. Für zwei gegebene LFSAs $A_1$ und $A_2$ ist es leicht einen Automaten zu konstruieren, welcher die Sprache $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ akzeptiert.

**Definition 4.23 (Synchrones Produkt von LFSAs)** *Seien $A_i = (\Sigma, S_i, S_i^0, \rho_i, F_i, l_i)$ für $i = 1, 2$ zwei LFSAs. Der* Produktautomat $A_1 \times A_2 = (\Sigma, S, S^0, \rho, F, l)$ *ist wie folgt definiert:*

- $S = \{(s_1, s_2) \in S_1 \times S_2 \,|\, l_1(s_1) = l_2(s_2)\}$
- $S^0 = (S_1^0 \times S_2^0) \cap S$
- $(s_1, s_2) \to (s_1', s_2')$ *gdw.* $s_1 \to s_1'$ *und* $s_2 \to s_2'$
- $F = (F_1 \times F_2) \cap S$
- $l(s_1, s_2) = l_1(s_1) = l_2(s_2)$

Zustände sind Paare $(s_1, s_2)$ mit $s_1 \in S_1$ und $s_2 \in S_2$, die die gleiche Markierung tragen, d.h. $l_1(s_1) = l_2(s_2)$. Es ist leicht zu sehen, dass $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ gilt, denn jeder akzeptierende Lauf $(s_0, s_0')(s_1, s_1') \ldots (s_n, s_n')$ von $A_1 \times A_2$ muss in einem Zustand $(s_n, s_n') \in F$ enden. Um aber $(s_n, s_n')$ zu erreichen, muss es möglich sein $s_n$ in $A_1$ über den Lauf $s_0 s_1 \ldots s_n$ zu erreichen, welches wiederum ein akzeptierender Lauf von $A_1$ ist, da $s_n \in F_1$. Analog muss $s_0' s_1' \ldots s_n'$ ein akzeptierender Lauf von $A_2$ sein, da $s_n' \in F_2$. Folglich ist die $i$-te Projektion (für $i = 1, 2$) jedes akzeptierendes Laufes von $A_1 \times A_2$ auch ein akzeptierender Lauf von $A_i$.

### Produktautomaten auf unendlichen Wörtern

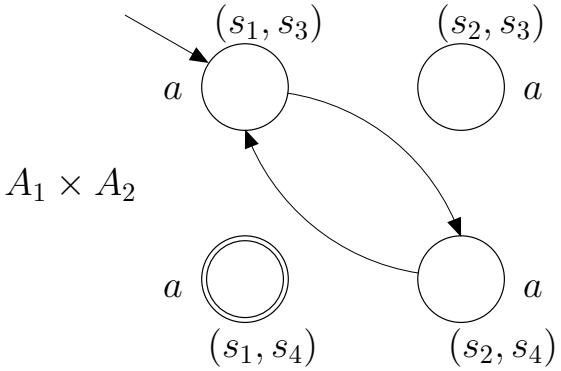
Für Büchi Automaten ist ein derart naives Verfahren leider ungeeignet. Gemäß obiger Konstruktionsvorschrift ist die Endzustandsmenge $F$ das Kreuzprodukt der beiden Endzustandsmengen $F_1$ und $F_2$. Betrachtet man $A_1$ und $A_2$ als Automaten auf unendlichen Wörtern so bedeutet dies, dass $A_1 \times A_2$ ein unendliches Wort $w$ akzeptiert, wenn zwei akzeptierende Läufe $\sigma_1$ und $\sigma_2$ für $A_1$ und $A_2$ existieren, welche beide unendlich oft und *gleichzeitig* akzeptierende Endzustände besuchen. Diese Anforderung ist zu streng und führt allgemein zu

$$\mathcal{L}_\omega(A_1 \times A_2) \quad \subseteq \quad \mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2),$$

was nicht dem gewünschten Ergebnis entspricht. Dies lässt sich an folgendem Beispiel veranschaulichen.

**Beispiel 4.11** *Seien $A_1$ und $A_2$ zwei Büchi Automaten, die wie folgt definiert sind:*


*Die von beiden LBAs akzeptierte Sprache ist $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2) = a^\omega$. Der Automat $A_1 \times A_2$ hingegen ist*




*und hat ganz offensichtlich keinen akzeptierenden Lauf. Mit anderen Worten ist $\mathcal{L}_\omega(A_1 \times A_2) = \emptyset$ im Gegensatz zu $\mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2) = a^\omega$. Das Problem resultiert aus der Tatsache, dass die Produktautomatkonstruktion von der Annahme ausgeht, $A_1$ und $A_2$ gingen zeitgleich durch den akzeptierenden Zustand, was allerdings in diesem Beispiel niemals der Fall ist, da $A_1$ und $A_2$ nach Konstruktion „phasenverschoben" sind: Wenn $A_1$ in einem akzeptierenden Zustand ist, ist $A_2$ nicht in einem akzeptierenden Zustand, und umgekehrt.*

*Betrachtet man hingegen die akzeptierten endlichen Wörter der Automaten, so ist*

$$\mathcal{L}(A_1) \;=\; \{a^{2n+1} \mid n \geq 0\}$$

*und*

$$\mathcal{L}(A_2) \;=\; \{a^{2n+2} \mid n \geq 0\},$$

*und insgesamt gilt*

$$\mathcal{L}(A_1 \times A_2) \;=\; \emptyset \;=\; \mathcal{L}(A_1) \cap \mathcal{L}(A_2).$$

Glücklicherweise existiert eine Modifikation der reinen Produktautomatenkonstruktion, die auch für Büchi Automaten funktioniert (vgl. [Cho74]).

**Definition 4.24 (Synchrones Produkt von LBAs)** *Seien $A_i = (\Sigma, S_i, S_i^0, \rho_i, F_i, l_i)$ für $i = 1, 2$ zwei LBAs. Der* Produktautomat *$A_1 \otimes A_2 = (\Sigma, S, S^0, \rho, F, l)$ ist wie folgt definiert:*

- $S = \{(s_1, s_2) \in S_1 \times S_2 \,|\, l_1(s_1) = l_2(s_2)\} \times 1, 2$
- $S^0 = \big((S_1^0 \times S_2^0) \times \{1\}\big) \cap S$
- *Wenn* $s_1 \to s_1'$ *und* $s_2 \to s_2'$, *dann*
  - *(i) wenn* $s_1 \in F_1$, *dann* $(s_1, s_2, 1) \to (s_1', s_2', 2)$
  - *(ii) wenn* $s_2 \in F_2$, *dann* $(s_1, s_2, 2) \to (s_1', s_2', 1)$
  - *(iii) anderenfalls* $(s_1, s_2, i) \to (s_1', s_2', i)$ *für* $i = 1, 2$
- $F = \big((F_1 \times S_2) \times \{1\}\big) \cap S$
- $l(s_1, s_2, i) = l_1(s_1) = l_2(s_2)$

Die intuitive Vorstellung hinter dieser Konstruktion ist die folgende: Der Automat $A = A_1 \otimes A_2$ führt simultan $A_1$ und $A_2$ auf der Eingabe aus. Entsprechend verhält sich $A$ als hätte er zwei „Berechnungseinheiten“, eine für $A_1$ und eine weitere für $A_2$. $A$ merkt sich nun nicht nur den Zustand der beiden Berechnungseinheiten (die ersten beiden Komponenten des Zustandstripels), sondern zusätzlich auch noch einen Zeiger, welcher auf eine der beiden Berechnungseinheiten zeigt (die dritte Komponenten des Zustands). Immer wenn nun eine Berechnungseinheit einen akzeptierenden Zustand durchläuft wechselt der Zeiger zur jeweils anderen Berechnungseinheit (Regel (i) und (ii) in der obigen Definition). Genauer gesagt, wenn ein akzeptierender Zustand von $A_i$ durchlaufen wird, wechselt der Zeiger zur Berechnungseinheit $(i + 1) \bmod 2$.

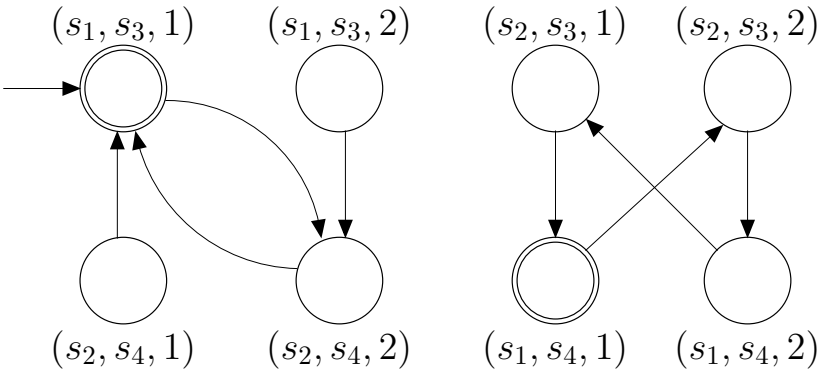
Das Akzeptanzkriterium garantiert, dass beide Berechnungseinheiten akzeptierende Zustände unendlich oft durchlaufen, da ein Lauf genau dann akzeptierend ist, wenn er unendlich oft $F_1 \times S_2 \times \{1\}$ durchläuft. Das wiederum bedeutet, dass $A_1$ einen akzeptierenden Zustand unendlich oft durchläuft während der Zeiger auf 1 steht. Da jedoch bei jedem Durchlaufen eines akzeptierenden Zustands von $A_1$ der Zeiger auf 2 wechselt, muss auch mindestens ein akzeptierender Zustand von $A_2$ unendlich oft durchlaufen werden, denn nur so kann der Zeiger wieder auf 1 wechseln. Um also einen akzeptierenden Zustand von $A_1 \otimes A_2$ unendlich oft zu durchlaufen, müssen $A_1$ und $A_2$ ihrerseits mindestens einen akzeptierenden Zustand unendlich oft durchlaufen. Folglich gilt

$$\mathcal{L}_\omega(A_1 \otimes A_2) \;=\; \mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2).$$

**Beispiel 4.12** *Seien* $A_1, A_2$ *wie in Beispiel 4.11. Der LBA* $A_1 \otimes A_2$ *wird wie folgt konstruiert: Die neue Zustandsmenge ist* $\{s_1, s_2\} \times \{s_3, s_4\} \times \{1, 2\}$, *also* $2^3 = 8$ *Zustände. Der Startzustand ist* $(s_1, s_3, 1)$. *Die akzeptierenden Zustände sind* $(s_1, s_3, 1)$ *und* $(s_1, s_4, 1)$. *Unter anderem lassen sich nach obiger Definition die folgenden Übergänge ableiten:*

$$\begin{array}{ll} (s_1, s_3, 1) \to (s_2, s_4, 2) & \text{da } s_1 \to s_2,\ s_3 \to s_4 \text{ und } s_1 \in F_1 \\ (s_1, s_4, 2) \to (s_2, s_3, 1) & \text{da } s_1 \to s_2,\ s_4 \to s_3 \text{ und } s_4 \in F_2 \\ (s_1, s_3, 2) \to (s_2, s_4, 1) & \text{da } s_1 \to s_2,\ s_3 \to s_4 \text{ und } s_3 \notin F_2 \end{array}$$

*Der erste Übergang folgt nach Regel (i), der zweite nach Regel (ii) und der dritte nach Regel (iii). Den Nachweis der verbleibenden Übergänge überlassen wir dem interessierten Leser. Der daraus resultierende Produktautomat $A_1 \otimes A_2$ sieht dann wie folgt aus*




*wobei alle Zustände mit a markiert sind. Offensichtlich ist die von diesem Automaten akzeptierte Sprache $a^\omega$, und entspricht damit $\mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2)$.*

Es sei noch darauf hingewiesen, dass die Anzahl der Zustände des Produktautomaten $A_1 \otimes A_2$ proportional zu $|S_1| \cdot |S_2|$ ist, wobei $S_1, S_2$ die Zustände von $A_1, A_2$ sind. Hierbei handelt es sich um eine *worst-case* Abschätzung; diese Obergrenze wird beim Model Checking üblicherweise nicht erreicht, da der Automat $A_{sys} \otimes A_{\neg\phi}$ in der Regel klein ist. Grund dafür ist die Tatsache, dass $A_{sys} \otimes A_{\neg\phi}$ nur Sequenzen akzeptiert, welche die Spezifikation $\phi$ verletzen, und von diesen erwartet man nur relativ wenige. Bei einem korrekten System existieren überhaupt keine derartigen Sequenzen.

### Das Leerheitsproblem

Gemäß Tabelle 4.5 ist nun das zweite Teilproblem zu lösen: Wie kann für einen gegebenen LBA $A$ geprüft werden, ob $A$ *leer* ist, d.h. ob $\mathcal{L}_\omega(A) = \emptyset$ gilt? Nach Definition ist ein LBA $A$ genau dann *nicht leer*, wenn von einem beliebigen Startzustand aus ein akzeptierender Zustand erreichbar ist, welcher zusätzlich von sich selbst aus erreichbar ist. Anders ausgedrückt, $A$ muss einen Zyklus enthalten, welcher zumindest einen akzeptierenden Zustand umfasst und von einem beliebigen Startzustand erreichbar ist. Hierbei ist ein Zustand $s'$ erreichbar von $s$, wenn eine Folge $s_0 \dots s_k$ existiert, so dass $s_0 = s$, $s_k = s'$ und $s_i \to s_{i+1}$ für alle $0 \leq i < k$ gilt.

**Satz 4.4** *Sei $A = (\Sigma, S, S^0, \rho, F, l)$ ein LBA. $\mathcal{L}_\omega(A) \neq \emptyset$ gdw. es existiert ein $s_0 \in S^0$ und ein $s' \in F$, so dass $s'$ sowohl von $s_0$ als auch $s'$ erreichbar ist.*

Den Beweis dieses Satzes überlassen wir dem Leser. Das Prinzip sollte anhand der vorangegangenen Erläuterung klar sein.

Um also zu bestimmen, ob ein gegebener Automat $A$ nicht leer ist, genügt es zu testen, ob ein erreichbarer Zyklus existiert, der einen akzeptierenden Zustand mit einschliesst.

Der entsprechende Algorithmus besteht aus zwei Schritten. Vereinfachend sei dazu im Folgenden angenommen, dass $A$ nur einen Startzustand $s_0$ besitzt, also $S^0 = \{s_0\}$.

1. Zunächst werden all akzeptierenden Zustände berechnet, die erreichbar sind von $s_0$. Zur weiteren Vereinfachung nehmen wir hierbei an, dass die akzeptierenden Zustände geordnet sind. Diese Berechnung erfolgt durch die Funktion *ReachAccept*, die in Listing 4.2 dargestellt ist.

2. Anschliessend wird geprüft, ob einer der durch *ReachAccept* berechneten Zustände Bestandteil eines Zyklus' ist. Dies wird durch die Funktion *DetectCycle* aus Listing 4.3 erledigt.

Das Hauptprogramm besteht dann aus **return** *DetectCycle*(*ReachAccept*($s_0$)).

```
function ReachAccept (s0: Vertex): sequence of Vertex;
(* Vorbedingung: keine *)
begin var S: sequence of Vertex, (* Pfad von s0 zum aktuellen s *)
          R: sequence of Vertex, (* erreichbare s ∈ F *)
          Z: set of Vertex; (* besuchte s *)
  S, R, Z := ⟨s0⟩, ⟨⟩, ∅;
  do S = ⟨s⟩⌢S' ⟶
     if ρ(s) ⊆ Z ⟶
        S := S';
        if s ∈ F ⟶ R := R⌢⟨s⟩
        [] s ∉ F ⟶ skip
        fi
     [] ρ(s) ⊈ Z ⟶
        let s' in ρ(s) \ Z;
        S, Z := ⟨s'⟩⌢S, Z ∪ {s'}
     fi
  od;
  return R;
end
(* Nachbedingung: R beinhaltet alle von s0 erreichbaren *)
(*  s ∈ F und zwar so geordnet, dass wenn R = ⟨s1, ..., sk⟩ *)
(* dann impliziert i < j, dass si vor sj erforscht wurde *)
```

Listing 4.2: Algorithmus zur Berechnung von erreichbaren akzeptierenden Zuständen

Der Algorithmus *ReachAccept* funktioniert wie folgt: Der Graph, der den Büchi Automaten repräsentiert, wird in einer Tiefensuche durchlaufen. Beginnend beim Startzustand $s_0$ wird in jedem Schritt ein neuer Zustand ausgewählt (sofern ein solcher existiert), der unmittelbar erreichbar ist vom aktuellen erforschten Knoten $s$. Wenn alle mit $s$ beginnenden Pfade erforscht sind, d.h. $\rho(s) \subseteq Z$, wird $s$ aus $S$ entfernt, und falls es sich

um einen akzeptierenden Zustand handelt, wird $s$ an $R$ angehängt. Der Algorithmus terminiert sobald alle möglichen mit $s_0$ beginnenden Pfade erforscht sind, d.h. wenn alle Zustände, die von $s_0$ erreichbar sind, besucht wurden. Interessant zu bemerken ist, dass alle Operationen auf $S$ sich ausschliesslich auf das erste Element beziehen; entsprechend bietet es sich an, $S$ als Stack zu implementieren.

```
function DetectCycle (R: sequence of Vertex): Bool;
(* Vorbedingung: keine *)
begin var S: sequence of Vertex,
          Z: set of Vertex,
          b: Bool;
  S, Z, b := ⟨⟩, ∅, false;
  do R = ⟨s⟩⌢R' ∧ ¬b ⟶
    R, S := R', ⟨s⟩⌢S;
    do S = ⟨s'⟩⌢S' ∧ ¬b ⟶
      b := (s ∈ ρ(s'));
      if ρ(s') ⊆ Z ⟶
        S := S'
      [] ρ(s') ⊈ Z ⟶
        let s'' in ρ(s') \ Z;
        S, Z := ⟨s''⟩⌢S, Z ∪ {s''}
      fi
    od
  od;
  return b;
end
(* Nachbedingung: b ist true, wenn ein Zustand aus R *)
(* Bestandteil eines Zyklus' ist, sonst ist b false *)
```

Listing 4.3: Algorithmus zum Test auf akzeptierenden Zyklus

Der Algorithmus *DetectCycle* führt ebenfalls eine Tiefensuche durch, beginnend mit den von $s_0$ erreichbaren akzeptierenden Zuständen. Er terminiert, wenn alle akzeptierenden Zustände in $R$ überprüft wurden oder wenn ein akzeptierender Zustand gefunden wurde, der Bestandteil eines Zyklus' ist. Dazu ist zu bemerken, dass die Elemente von $R$ immer von vorne entfernt werden, während *ReachAccept* Element immer am Ende von $R$ eingefügt. Es empfiehlt sich aus diesem Grund, $R$ als *First In First Out* Queue (FIFO) zu implementieren.

Statt nun *ReachAccept* und *DetectCycle* in zwei Phasen zu berechnen, ist es möglich für einen akzeptierenden Zustand unmittelbar zu prüfen, ob dieser Bestandteil eines Zyklus' ist, und zwar während die von $s_0$ erreichbaren akzeptierenden Zustände bestimmt werden. Dies lässt sich durch eine ineinander geschachtelte Tiefensuche realisieren: Die äussere Suche identifiziert akzeptierende Zustände, die erreichbar sind von $s_0$, während

die innere Suche überprüft, ob ein solcher Zustand Bestandteil eines Zyklus' ist. Das resultierende Programm ist in Listing 4.4 dargestellt.

Angenommen dieser Algorithmus wird nun auf den Produktautomaten $A_{sys} \otimes A_{\neg\phi}$ angewandt. Ein interessanter Aspekt des Programms ist, dass im Falle eines Fehlers – also eines Zyklus' innerhalb von $A_{sys} \otimes A_{\neg\phi}$, welcher den akzeptierenden Zustand $s$ enthält – auf einfache Weise ein Beispiel für einen inkorrekten Pfad berechnet werden kann: $S_1$ enthält dann den Pfad von $s_0$ nach $s$ (in umgekehrter Reihenfolge), während $S_2$ den Zyklus von $s$ nach $s$ (wiederum in umgekehrter Reihenfolge) beschreibt. Auf diese Weise kann ein Gegenbeispiel, das aufzeigt wie $\phi$ verletzt wird, konstruiert werden, nämlich $(S_1' \frown S_2)^{-1}$ mit $S_1 = \langle s \rangle \frown S_1'$.

Die Zeitkomplexität von *ReachAcceptAndDetectCycle* ist proportional zur Anzahl der Zustände plus der Grösse der Übergangsrelation des LBA, also $\mathcal{O}(|S| + |\rho|)$. Allerdings wurde bei der Beschreibung des Algorithmus die Startzustandsmenge $S_0$ als einelementig angenommen. Allgemein ergibt sich daraus, dass die gesamte Prozedur für alle Startzustände durchgeführt werden muss. Dann ergibt sich eine *worst-case* Laufzeitkomplexität von $\mathcal{O}\big(|S^0| \cdot (|S| + |\rho|)\big)$.

### 4.4.5. Zusammenfassung

Die einzelnen Schritte des LTL Model Checking Algorithmus werden wie folgt kombiniert: Zunächst wird das Modell des Systems *sys* in einen Büchi Automaten $A_{sys}$ umgewandelt, in dem alle Zustände akzeptierend sind. Die Spezifikation wird als LTL Formel $\phi$ formuliert, negiert und in einen zweiten Büchi Automaten $A_{\neg\phi}$ überführt. Das Produkt dieser beiden Automaten repräsentiert alle möglichen Berechnungen, welche die Eigenschaft $\phi$ verletzen. Durch einen Test auf Leerheit dieses Automaten wird bestimmt, ob $\phi$ durch das Modell des Systems *sys* erfüllt wird oder nicht.

Obwohl die Schritte des Algorithmus in einer strengen Reihenfolge vorgestellt wurden, die dadurch zunächst den Anschein erweckt, dass der nächste Schritt nicht durchgeführt werden, bevor der vorherige vollständig abgeschlossen ist, können einige Schritte *„on-the-fly“* durchgeführt werden. Beispielsweise kann die Konstruktion des Graphen für eine LTL-Formel in Normalform durchgeführt werden, während der Produktautomat $A_{sys} \otimes A_{\neg\phi}$ auf Leerheit geprüft wird. Hierbei wird der Graph *„on-demand“* konstruiert: Ein neuer Knoten wird erst dann benötigt, wenn bis dahin im teilweise konstruierten Produktautomaten kein akzeptierender Zyklus gefunden wurde. Bei der Konstruktion der Nachfolger eines Knotens im Graphen, konstruiert man zunächst nur diejenigen, welche dem aktuellen Zustand des Automaten $A_{sys}$ entsprechen, anstatt alle möglichen Nachfolger. Dadurch ist es möglich einen akzeptierenden Zyklus zu finden, ohne zunächst den vollständigen Graphen $\mathcal{G}_{\neg\phi}$ konstruieren zu müssen.

In ähnlicher Weise muss auch der Automat $A_{sys}$ nicht vollständig verfügbar sein bevor mit der Prüfung auf Leerheit des Produktautomaten begonnen werden kann. Dies ist im Allgemeinen der entscheidendste Vorteil des *on-the-fly Model Checkings*, da der Automat für das System üblicherweise sehr groß ist. Indem man die vollständige Betrachtung des

```
function ReachAcceptAndDetectCycle (s_0: Vertex): Bool;
begin var S_1, S_2: sequence of Vertex,
          Z_1, Z_2: set of Vertex,
          b: Bool;
  S_1, S_2, Z_1, Z_2, b := ⟨s_0⟩, ⟨⟩, ∅, ∅, false;
  do S_1 = ⟨s⟩⌢S'_1 ∧ ¬b ⟶
    if ρ(s) ⊆ Z_1 ⟶
      S_1 := S'_1;
      if s ∉ F ⟶
        skip
      [] s ∈ F ⟶
        S_2 := ⟨s⟩⌢S_2;
        do S_2 = ⟨s'⟩⌢S'_2 ∧ ¬b ⟶
          b := (s ∈ ρ(s'));
          if ρ(s') ⊆ Z_2 ⟶
            S_2 := S'_2
          [] ρ(s') ⊈ Z_2 ⟶
            let s'' in ρ(s') \ Z_2;
            S_2, Z_2 := ⟨s''⟩⌢S'_2, Z_2 ∪ {s''}
          fi
        od
      fi
    [] ρ(s) ⊈ Z_1 ⟶
      let s'' in ρ(s) \ Z_1;
      S_1, Z_1 := ⟨s''⟩⌢S'_1, Z_1 ∪ {s''}
    fi
  od;
  return b;
end
```

Listing 4.4: Kombiniertes Programm für *ReachAccept* und *DetectCycle*

Systems vermeidet, reduziert sich der Speicherbedarf des Verfahrens signifikant.

Abschliessend betrachten wir noch einmal zusammenfassend die *worst-case* Laufzeitkomplexität des Model Checking Algorithmus für LTL-Formeln. Sei $\phi$ eine als LTL-Formel kodierte Spezifikation und *sys* das Modell des Systems, welches verifiziert werden soll. Der entscheidende Schritt ist die Konstruktion eines Graphen zu einer LTL-Formel in Normalform. Da jeder Knoten im Graphen mit einer Menge von Teilformeln von $\neg\phi$ markiert ist, ist die Knotenanzahl proportional zur Anzahl der Mengen von Teilformeln von $\neg\phi$, also $\mathcal{O}(2^{|\neg\phi|})$.

Da die weiteren Schritte der Transformation von $\neg\phi$ in einen LBA diese *worst-case* Komplexität nicht beeinflussen, hat der resultierende LBA einen Speicherplatzbedarf von $\mathcal{O}(2^{|\neg\phi|})$. Der *worst-case* Speicherbedarf des Produktautomaten $A_{sys} \otimes A_{\neg\phi}$ ist dementsprechend $\mathcal{O}(|S_{sys}| \cdot 2^{|\neg\phi|})$, wobei $S_{sys}$ die Zustandsmenge von $A_{sys}$ ist. Da die Laufzeitkomplexität von *Checking for emptiness* direkt proportional zur Anzahl der Zustände und Grösse der Übergangsrelation ist, und da schlimmstensfalls $|S_{sys}|^2$ Übergänge im Produktautomaten existieren, erhalten wir

$$\mathcal{O}(|S_{sys}|^2 \cdot 2^{|\neg\phi|})$$

als *worst-case* Laufzeitkomplexität für den LTL Model Checking Algorithmus. In der Literatur findet man häufig die Aussage, dass die Zeitkomplexität des Algorithmus linear in der Größe des Systems (anstatt quadratisch) und exponentiell in der Größe der Spezifikation ist.

Die Tatsache, dass die Laufzeitkomplexität exponentiell in der Länge der LTL Formel ist, erscheint zunächst eine erhebliche Hürde für die Adaption von LTL Model Checking für praktische Systeme. Experimente haben jedoch gezeigt, dass diese Abhängigkeit nicht signifikant ist, da die Spezifikation typischerweise sehr kurz ist, wie auch in Abschnitt 4.3.3 gut zu sehen. Für praktische Anwendungen wird der Faktor $2^{|\neg\phi|}$ deshalb häufig als konstant angenommen, und man sagt, die Komplexität ist quadratisch in der Größe des Systems.

## 4.5. Branching Time Logic

In der Beschreibung der LTL (Linear Time Logic) haben wir gesehen, dass die Semantik von LTL-Formeln über *Pfade* definiert ist (siehe dazu Definition 4.5). Und zwar wurde definiert, dass ein Zustand eines Systems eine LTL-Formel erfüllt, wenn sie auf *allen Pfaden* beginnend von diesem Zustand erfüllt ist. Ausgehend von dieser Beobachtung stellen wir fest, dass Eigenschaften, welche die Existenz eines Pfades erfordern, in LTL nicht formuliert werden können. Teilweise ist es möglich dieses Problem zu umgehend, indem man die Negation der Eigenschaft betrachtet und das Ergebnis entsprechend interpretiert. Um also zu prüfen, ob ein Pfad von einem Zustand $s$ existiert, welcher die LTL-Formel $\phi$ erfüllt, prüft man dann ob alle von $s$ ausgehenden Pfade die Formel $\neg\phi$ erfüllen; eine positive Antwort ist dann eine negative Antwort auf die ursprüngliche

Fragestellung. Allerdings ist es in LTL unmöglich, Eigenschaften zu formulieren, die sowohl universelle als auch existentielle Pfadquantorisierung erfordern.

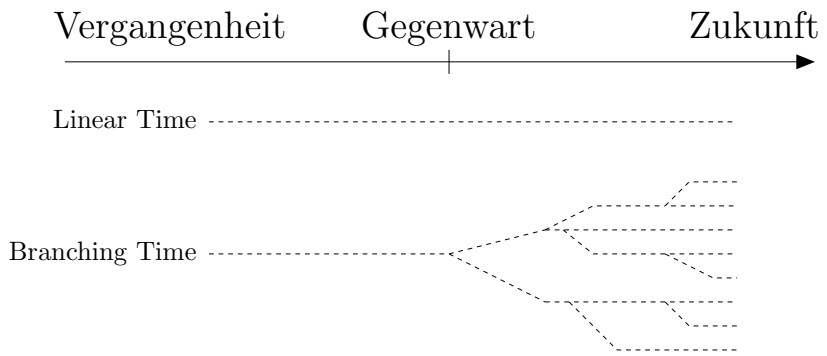



Abbildung 4.3.: Zeitmodelle

Branching Time Logiken lösen dieses Problem indem ermöglicht wird, explizit über Pfade zu quantorisieren. Dies impliziert insbesondere, dass statt eines linearen Zeitmodells, wie es zur Definition der Semantik von linearen Temporallogiken gewählt wird, ein Zeitmodell benötigt wird, welches Verzweigungen in der Zukunft erlaubt. Dieser Unterschied ist in Abbildung 4.3 dargestellt.

Wir betrachten im Folgenden eine solche Temporallogik, die *Computational Tree Logic* (CTL). In CTL existieren neben den bereits aus der LTL bekannten temporalen Operatoren **X**, **G**, **F** und **U** zwei Quantoren **A** und **E**, mit der Bedeutung „für alle Pfade“ bzw. „es existiert ein Pfad“. Beispielsweise können wir schreiben:

- Es existiert ein erreichbarer Zustand, der $\phi$ erfüllt: $\mathbf{EF}\,\phi$.
- Es existiert ein erreichbarer Zustand, von dem aus alle erreichbaren Zustände $\phi$ erfüllen: $\mathbf{EF}\,\mathbf{AG}\,\phi$.
- Von allen erreichbaren Zuständen, die $\phi$ erfüllen, ist es möglich, dass $\phi$ solange erfüllt bleibt, bis ein Zustand erreicht ist, der $\psi$ erfüllt: $\mathbf{AG}\,\big(\phi \rightarrow \mathbf{E}[\phi\,\mathbf{U}\,\psi]\big)$.
- Sobald ein Zustand erreicht ist, der $\phi$ erfüllt, ist es möglich, dass $\phi$ von da an „für alle Zeiten“ gilt: $\mathbf{AG}\,\big(\phi \rightarrow \mathbf{EG}\,\phi\big)$.

Wie sich der Leser leicht überzeugen kann, ist es unmöglich, diese Eigenschaften (direkt) in LTL zu formulieren. Der Grund hierfür ist einfach, dass LTL-Formeln implizit über alle Pfade quantifizieren, was nicht zuletzt darin begründet ist, dass in einem LTL-Modell gemäß Definition 4.4 zu jedem Zustand nur genau ein Folgezustand existiert[10].

[10]Es sei an dieser Stelle noch angemerkt, dass neben dieser Definition noch andere Definitionen für die Semantik von LTL existieren, die jedoch im Ergebnis alle den gleichen Einschränkungen unterliegen.

### 4.5.1. Syntax von CTL

Die Syntax der Computational Tree Logic wird wie folgt definiert. Wie zuvor im Fall der LTL, bilden die Propositionalzeichen die elementaren Bestandteile von CTL-Formeln, und die Aussagenlogik ist eine echte Teilmenge der CTL.

**Definition 4.25 (Syntax von CTL)** *Sei $\mathcal{AP}$ die Menge von Propositionalzeichen. Die Menge $\mathcal{L}_{CTL}$ der CTL-Formeln ist wie folgt induktiv definiert:*

1. *Jedes Propositionalzeichen $p \in \mathcal{AP}$ ist eine Formel.*
2. *Sind $\phi, \psi \in \mathcal{L}_{CTL}$ Formeln, so sind auch $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$ und $\phi \rightarrow \psi$ Formeln.*
3. *Sind $\phi, \psi \in \mathcal{L}_{CTL}$ Formlen, so sind auch $\mathbf{AX}\,\phi$, $\mathbf{AF}\,\phi$, $\mathbf{AG}\,\phi$ und $\mathbf{A}[\phi\,\mathbf{U}\,\psi]$ Formeln.*
4. *Sind $\phi, \psi \in \mathcal{L}_{CTL}$ Formlen, so sind auch $\mathbf{EX}\,\phi$, $\mathbf{EF}\,\phi$, $\mathbf{EG}\,\phi$ und $\mathbf{E}[\phi\,\mathbf{U}\,\psi]$ Formeln.*

Die temporalen Verknüpfungen der CTL treten stets als Symbolpaar auf, wobei das erste Zeichen stets **A** oder **E** ist. **A** bedeutet „entlang aller Pfade“ (unvermeidbar) und **E** bedeutet „entlang eines Pfades“ oder „es existiert ein Pfad“ (möglicherweise). Das zweite Element des Paars ist entweder **X**, **G**, **F** oder **U**, mit den bereits bekannten Bedeutungen „nächster Zustand (NeXt)“, „alle zukünftigen Zustände (Globally)“, „ein zukünftiger Zustand (Future)“ und „bis zu einem zukünftigen Zustand (Until)“. Das Symbolpaar in $\mathbf{A}[\phi\,\mathbf{U}\,\psi]$ ist beispielsweise **AU**. Die Symbole **X**, **G**, **F** und **U** dürfen niemals alleine auftreten, d.h. sie müssen stets mit **A** oder **E** kombiniert sein. Entsprechend dürfen auch die Symbole **A** und **E** nur in Kombination mit **X**, **G**, **F** oder **U** auftreten.

### 4.5.2. Semantik von CTL

CTL-Formeln werden, ebenso wie LTL-Formeln, über Transitionssystemen interpretiert (siehe dazu Definition 4.2). Entsprechend definieren wir:

**Definition 4.26 (CTL-Modell)** *Jedes Transitionssystem $\mathcal{M} = (S, \rightarrow, l)$ ist ein* CTL-Modell.

Sei $\mathcal{M} = (S, \rightarrow, l)$ ein solches Modell, $s \in S$ ein Zustand des Modells und $\phi \in \mathcal{L}_{CTL}$ eine CTL-Formel. Ob $\mathcal{M}, s \models \phi$ gilt, wird wieder wie üblich induktiv über die Struktur von $\phi$ definiert, und kann informal etwa wie folgt verstanden werden:

- Ist $\phi$ atomar, so wird die Erfüllbarkeit durch $l$ bestimmt.
- Besteht $\phi$ aus einer aussagenlogischen Verknüpfung von Formeln, so wird die Erfüllbarkeit über die üblichen Wahrheitstafeln und die Wahrheitswerte der Teilformeln bestimmt.

- Besteht $\phi$ hingegen aus einer temporalen Verknüpfung von Formeln mit einem der **A**-Operatoren, so ist die Formel erfüllbar, wenn *alle* in $s$ beginnenden Pfade die „LTL Formel“ $\phi'$ erfüllen, die aus $\phi$ entsteht, indem das **A** entfernt wird.
- Analog für die **E**-Operatoren.

Dies entspricht natürlich nur der intuitiven Vorstellung. Durch das Entfernen von **A** resp. **E** entsteht nicht zwangsläufig eine gültige LTL-Formel, da diese ja weitere Vorkommen von **A** und **E** enthalten kann. Formal korrekt lässt sich die Erfüllbarkeit für CTL-Formeln wie folgt definieren.

**Definition 4.27** *Sei $\mathcal{M} = (S, \to, l)$ ein Modell und $s \in S$ ein Zustand in $\mathcal{M}$. Die Menge aller Pfade $\pi$ von $\mathcal{M}$ die im Zustand $s$ beginnen ist durch*

$$\Pi_{\mathcal{M}}(s) \quad = \quad \{\pi \in S^{\omega} \mid \pi(1) = s\}$$

*definiert.*

**Definition 4.28** *Sei $\mathcal{M} = (S, \to, l)$ ein CTL-Modell, $s \in S$ ein Zustand und seien $\phi, \psi \in \mathcal{L}_{CTL}$ Formeln. Die Erfüllbarkeitsrelation $\mathcal{M}, s \models \phi$ ist wie folgt induktiv über die Struktur von CTL Formeln definiert:*

1. $\mathcal{M}, s \models p$ *gdw.* $p \in \mathcal{AP}$
2. $\mathcal{M}, s \models \neg\phi$ *gdw.* $\mathcal{M}, s \not\models \phi$
3. $\mathcal{M}, s \models \phi \vee \psi$ *gdw.* $\mathcal{M}, s \models \phi$ *oder* $\mathcal{M}, s \models \psi$
4. $\mathcal{M}, s \models \phi \wedge \psi$ *gdw.* $\mathcal{M}, s \models \phi$ *und* $\mathcal{M}, s \models \psi$
5. $\mathcal{M}, s \models \phi \to \psi$ *gdw.* $\mathcal{M}, s \not\models \phi$ *oder* $\mathcal{M}, s \models \psi$
6. $\mathcal{M}, s \models \mathbf{AX}\,\phi$ *gdw.* $\mathcal{M}, \pi(2) \models \phi$ *für alle* $\pi \in \Pi_{\mathcal{M}}(s)$
7. $\mathcal{M}, s \models \mathbf{EX}\,\phi$ *gdw. es existiert ein* $\pi \in \Pi_{\mathcal{M}}(s)$ *so dass* $\mathcal{M}, \pi(2) \models \phi$
8. $\mathcal{M}, s \models \mathbf{AG}\,\phi$ *gdw.* $\mathcal{M}, \pi(i) \models \phi$ *für alle* $\pi \in \Pi_{\mathcal{M}}(s)$ *und* $i \geq 1$
9. $\mathcal{M}, s \models \mathbf{EG}\,\phi$ *gdw. es existiert ein* $\pi \in \Pi_{\mathcal{M}}(s)$ *so dass* $\mathcal{M}, \pi(i) \models \phi$ *für alle* $i \geq 1$
10. $\mathcal{M}, s \models \mathbf{AF}\,\phi$ *gdw. für alle* $\pi \in \Pi_{\mathcal{M}}(s)$ *existiert ein* $i \geq 1$ *mit* $\mathcal{M}, \pi(i) \models \phi$
11. $\mathcal{M}, s \models \mathbf{EF}\,\phi$ *gdw. es existiert ein* $\pi \in \Pi_{\mathcal{M}}(s)$ *und ein* $i \geq 1$ *mit* $\mathcal{M}, \pi(i) \models \phi$
12. $\mathcal{M}, s \models \mathbf{A}[\phi\,\mathbf{U}\,\psi]$ *gdw. für alle* $\pi \in \Pi_{\mathcal{M}}(s)$ *existiert ein* $i \geq 1$ *mit* $\mathcal{M}, \pi(i) \models \psi$ *und* $\mathcal{M}, \pi(j) \models \phi$ *für alle* $1 \leq j < i$

13. $\mathcal{M}, s \models \mathbf{E}[\phi \,\mathbf{U}\, \psi]$ *gdw. es existiert ein* $\pi \in \Pi_{\mathcal{M}}(s)$ *und ein* $i \geq 1$ *mit* $\mathcal{M}, \pi(i) \models \psi$ *und* $\mathcal{M}, \pi(j) \models \phi$ *für alle* $1 \leq j < i$

Geht $\mathcal{M}$ aus dem Zusammenhang hervor, so schreiben wir kurz $s \models \phi$ statt $\mathcal{M}, s \models \phi$. Man sagt in diesem Fall wieder $\phi$ *gilt in* $s$ oder $s$ *erfüllt* $\phi$. Analog zur Semantik der LTL definieren wir weiter:

**Definition 4.29 (Semantische Äquivalenz)** *Zwei CTL-Formeln* $\phi, \psi \in \mathcal{L}_{CTL}$ *heissen* semantisch äquivalent*, oder einfach* äquivalent*, geschrieben* $\phi \equiv \psi$*, wenn für alle Modelle* $\mathcal{M} = (S, \to, l)$ *und alle Zustände* $s \in S$ *gilt:* $s \models \phi$ *gdw.* $s \models \psi$*.*

Wie bereits erläutert, ist **A** ein Allquantor über Pfade und **E** der entsprechende Existenzquantor. Weiter handelt es sich bei **G** und **F** ebenfalls um All- und Existenzquantoren, welche über die Zustände auf einem bestimmten Pfad quantifizieren. Es dürfte demzufolge wenig überraschen, dass Äquivalenzen existieren, die an die De Morgan'schen Regeln erinnern:

$$\begin{aligned} \neg\mathbf{AF}\,\phi &\equiv \mathbf{EG}\,\neg\phi \\ \neg\mathbf{EF}\,\phi &\equiv \mathbf{AG}\,\neg\phi \\ \neg\mathbf{AX}\,\phi &\equiv \mathbf{EX}\,\neg\phi \end{aligned}$$

Daneben gelten die Äquivalenzen

$$\begin{aligned} \mathbf{AF}\,\phi &\equiv \mathbf{A}[\top \,\mathbf{U}\, \phi] \\ \mathbf{EF}\,\phi &\equiv \mathbf{E}[\top \,\mathbf{U}\, \phi], \end{aligned}$$

welche an die aus der LTL bekannten Äquivalenzen erinnern. Bemerkenswert sind auch die folgenden Äquivalenzen:

$$\begin{aligned} \mathbf{AF}\,\phi &\equiv \phi \vee \mathbf{AX}\,\mathbf{AF}\,\phi \\ \mathbf{EF}\,\phi &\equiv \phi \vee \mathbf{EX}\,\mathbf{EF}\,\phi \\ \mathbf{AG}\,\phi &\equiv \phi \wedge \mathbf{AX}\,\mathbf{AG}\,\phi \\ \mathbf{EG}\,\phi &\equiv \phi \wedge \mathbf{EX}\,\mathbf{EG}\,\phi \\ \mathbf{A}[\phi \,\mathbf{U}\, \psi] &\equiv \psi \vee \big(\phi \wedge \mathbf{AX}\,\mathbf{A}[\phi \,\mathbf{U}\, \psi]\big) \\ \mathbf{E}[\phi \,\mathbf{U}\, \psi] &\equiv \psi \vee \big(\phi \wedge \mathbf{EX}\,\mathbf{E}[\phi \,\mathbf{U}\, \psi]\big) \end{aligned}$$

Beispielsweise lässt sich die erste intuitiv wie folgt interpretieren: Damit $\mathbf{AF}\,\phi$ in einem bestimmten Zustand gilt, muss $\phi$ auf allen Pfaden ausgehend von diesem Zustand irgendwann einmal gelten. Dies ist aber genau dann der Fall, wenn $\phi$ entweder bereits in diesem Zustand gilt, oder die Formel $\mathbf{AF}\,\phi$ gilt für alle unmittelbaren Folgezustände.

**Definition 4.30 (Adäquate Mengen)** *Eine Menge* $L \subseteq \mathcal{L}_{CTL}$ *von CTL-Formeln heißt* adäquat*, wenn für alle* $\phi \in \mathcal{L}_{CTL}$ *ein* $\psi \in L$ *existiert, so dass gilt:* $\phi \equiv \psi$*.*

Eine Menge $L$ von CTL-Formeln ist also genau dann *adäquat*, wenn sich zu jeder beliebigen CTL-Formel $\phi$ eine Formel $\psi \in L$ finden lässt, für die $\phi \equiv \psi$ gilt, welche also die gleiche Bedeutung wie $\phi$ hat. Allgemein gilt für adäquate Mengen von CTL-Formeln der nachfolgende Satz.

**Satz 4.5 (Adäquate Mengen)** *Sei $L \subseteq \mathcal{L}_{CTL}$ eine Menge von CTL-Formeln und $\phi, \psi \in L$. Wenn $L$ adäquat ist, dann gilt:*

- $(\mathbf{E}[\phi \,\mathbf{U}\, \psi]) \in L$
- $(\mathbf{AX}\, \phi) \in L$ *oder* $(\mathbf{EX}\, \phi) \in L$
- $(\mathbf{EG}\, \phi) \in L$, $(\mathbf{AF}\, \phi) \in L$ *oder* $(\mathbf{A}[\phi \,\mathbf{U}\, \psi]) \in L$

**Beweis:** Der Beweis dieses Satzes findet sich in [Mar01]. □

Damit lässt sich nun – ähnlich wie zuvor für LTL – eine Normalform für CTL-Formeln definieren.

**Definition 4.31 (Normalform von CTL-Formeln)** *Die Menge $\mathcal{L}_{CTL}^{Norm} \subseteq \mathcal{L}_{CTL}$ ist wie folgt induktiv definiert:*

1. *Jedes $p \in \mathcal{AP}$ ist eine CTL-Formel in Normalform.*
2. *Sind $\phi, \psi$ CTL-Formeln in Normalform, so auch $\neg\phi$, $\phi \vee \psi$, $\mathbf{EX}\, \phi$, $\mathbf{E}[\phi \,\mathbf{U}\, \psi]$ und $\mathbf{AF}\, \phi$.*

**Lemma 4.2** *Die Menge $\mathcal{L}_{CTL}^{Norm}$ ist* adäquat.

**Beweis:** Bleibt als Übung für den Leser. □

### 4.5.3. CTL Spezifikationen

Zum besseren Verständnis der CTL betrachten wir in diesem Abschnitt die Spezifikation von Eigenschaften eines einfachen Systems zur Regelung des beiderseitigen Ausschlusses (engl.: *mutual exclusion*) zwischen zwei kommunizierenden Prozessen. Die beiden Prozesse $P_1$, $P_2$ können sich dabei jeweils in einem der folgenden drei Abschnitte befinden: Der kritische Abschnitt ($C$), der wartende Abschnitt ($T$) und der unkritische Abschnitt ($N$).

Prozesse starten im unkritischen Abschnitt und vor Eintritt in den kritischen Abschnitt muss zunächst der wartenden Abschnitt betreten werden. In diesem verweilt der Prozess bis er Zugang zum kritischen Abschnitt bekommt, und nach Abschluss des kritischen Abschnitts wechselt der Prozess zurück in den unkritischen Abschnitt. Der Zustand des Prozesses $P_i$ wird mit $P_i.s$ beschrieben, für $i = 1, 2$. Einige der interessanten Eigenschaften des Systems sind im Folgenden aufgelistet:

- „Es ist zu keinem Zeitpunkt möglich, dass sich beide Prozesse im kritischen Abschnitt befinden“.

$$\mathbf{AG}\, \neg(P_1.s = C \wedge P_2.s = C)$$

- „Kein Prozess wartet ewig“.

$$\mathbf{AG}\left[P_i.s = T \rightarrow \mathbf{AF}\,(P_i.s = C)\right]$$

- „Prozesse müssen abwechselnd Zugang zum kritischen Abschnitt erhalten“.

$$\mathbf{AG}\left[P_1.s = C \rightarrow \mathbf{A}[P_1.s = C\,\mathbf{U}\,(P_1.s \neq C \wedge \mathbf{A}[P_1.s \neq C\,\mathbf{U}\,P_2.s = C])]\right]$$

Die erste ist die wohl entscheidenste Eigenschaft, denn sie stellt sicher, dass das System korrekt arbeitet. Die zweite Eigenschaft wird üblicherweise als *Lifeness* bezeichnet und stellt sicher, dass das System keinen *Deadlock* enthält. Die letzte Eigenschaft sichert die *Fairness* innerhalb des Systems, wird jedoch häufig in einer schwächeren Form verwendet.

# 4.6. CTL Model Checking

In Abschnitt 4.4 wurde ein auf Büchi Automaten basierter Model Checking Algorithmus für LTL vorgestellt. Ermöglicht wird diese Vorgehensweise durch die Tatsache, dass zu jeder LTL-Formel $\phi$ ein Büchi Automat konstruiert werden kann, welcher exakt die (unendlichen) Folgen von Propositionalzeichen akzeptiert, für die $\phi$ gilt. Auf diese Weise lässt sich das LTL Model Checking Problem auf bekannte automatentheoretische Probleme reduzieren.

Damit stellt sich nun die Frage, ob für CTL ein ähnliches Vorgehen möglich ist. Tatsächlich ist es möglich das CTL Model Checking Problem ebenfalls auf automatentheoretische Probleme zu reduzieren. Im Unterschied zu LTL, wo Eigenschaften auf unendliche Folgen von Zuständen bezogen sind, beziehen sich Eigenschaften bei CTL auf *unendliche Bäume* von Zuständen. Entsprechend müssen für CTL-Formeln sogenannte *Büchi-tree automata* konstruiert werden. Die Zeitkomplexität dieser Konstruktion ist *exponentiell* in der Länge der CTL-Formel, analog zum Fall der LTL.

Obwohl auf dem Gebiet der automatenbasierten Verfahren zum CTL Model Checking in den letzten Jahren interessante Entwicklungen stattgefunden haben, werden wir im Rahmen dieses Dokuments nicht weiter darauf eingehen. Stattdessen wird das ursprünglich von Clarke und Emerson [CE82] vorgeschlagene Verfahren zum CTL Model Checking vorgestellt, welches auf Ergebnissen der *Bereichstheorie* (engl.: *domain theory*), insbesondere der *Fixpunkttheorie* (engl.: *fixed point theory*), basiert. Die nachfolgende Beschreibung des Algorithmus basiert im Wesentlichen auf den Erläuterungen in [HR04] und [Kat99].

## 4.6.1. Algorithmus

Die Aufgabe des Algorithmus besteht darin zu entscheiden, ob eine CTL-Formel[11] $\phi \in \mathcal{L}_{CTL}^{Norm}$ in einem endlichen CTL-Modell $\mathcal{M} = (S, \rightarrow, l)$ gültig ist. Die grundlegende Idee

[11] In diesem Abschnitt wird davon ausgegangen, dass CTL-Formeln grundsätzlich in Normalform vorliegen.

des Model Checking Algorithmus ist nun, jeden Zustand $s \in S$ mit denjenigen Teilformeln von $\phi$ zu *markieren*, die in $s$ gelten. Die Menge aller Teilformeln von $\phi$ wird mit $Sub(\phi)$ bezeichnet und ist wie folgt definiert.

**Definition 4.32 (Teilformeln von CTL-Formeln)** *Seien $p \in \mathcal{AP}$ und $\phi, \psi \in \mathcal{L}_{CTL}^{Norm}$. Die Menge aller Teilformeln einer CTL-Formel ist wie folgt induktiv definiert:*

- $Sub(p) = \{p\}$
- $Sub(\neg\phi) = Sub(\phi) \cup \{\neg\phi\}$
- $Sub(\phi \vee \psi) = Sub(\phi) \cup Sub(\psi) \cup \{\phi \vee \psi\}$
- $Sub(\mathbf{EX}\,\phi) = Sub(\phi) \cup \{\mathbf{EX}\,\phi\}$
- $Sub(\mathbf{AF}\,\phi) = Sub(\phi) \cup \{\mathbf{AF}\,\phi\}$
- $Sub(\mathbf{E}[\phi\,\mathbf{U}\,\psi]) = Sub(\phi) \cup Sub(\psi) \cup \{\mathbf{E}[\phi\,\mathbf{U}\,\psi]\}$

Die angesprochene Markierung der Zustände kann nun iterativ erfolgen. Zunächst werden dabei alle Zustände mit den Teilformeln der Länge 1, also den Propositionalzeichen, die in $\phi$ vorkommen markiert. In der $(n+1)$-ten Iteration des Markierungsalgorithmus werden dann die Teilformeln der Länge $(n+1)$ betrachtet und die Zustände entsprechend markiert. Beispielsweise werden alle Zustände $s$ mit der Formel $\phi \vee \psi$ markiert, die zuvor bereits mit $\phi$ oder $\psi$ markiert worden sind.

Das Model Checking Problem für CTL, zu entscheiden, ob $\mathcal{M}, s \models \phi$ für ein gegebenes CTL-Modell $\mathcal{M} = (S, \to, l)$ und eine CTL-Formel $\phi$ gilt, kann nun für ein beliebiges $s \in S$ gelöst werden, indem man die *Markierungen* von $s$ betrachtet:

$$\mathcal{M}, s \models \phi \quad :\Leftrightarrow \quad s \text{ ist mit } \phi \textit{ markiert.}$$

Tatsächlich lässt sich dieses Verfahren auf eine sehr elegante und kompakte Weise umsetzen, indem für ein gegebenes Modell $\mathcal{M}$ und eine gegebene Formel $\phi$

$$Sat(\phi) \quad := \quad \{s \in S \mid \mathcal{M}, s \models \phi\}$$

auf iterative Weise (wie oben angedeutet) berechnet. Das Problem $\mathcal{M}, s \models \phi$ reduziert sich dann auf $s \in Sat(\phi)$, wobei $Sat(\phi)$ durch den in Listing 4.5 angegebenen Algorithmus berechnet werden kann.

Offensichtlich wird durch die Berechnung von $Sat(\phi)$ ein allgemeineres Problem gelöst, als *nur* zu prüfen ob $\mathcal{M}, s \models \phi$ für einen Zustand $s$ gilt. Tatsächlich wird für *jeden* Zustand $s \in S$ berechnet, ob $\mathcal{M}, s \models \phi$ gilt. Zusätzlich werden, da $Sat(\phi)$ auf rekursive Weise über die Teilformeln von $\phi$ berechnet wird, auch die Mengen $Sat(\psi)$ für jede Teilformel $\psi \in Sub(\phi)$ berechnet, es könnte also auch unmittelbar überprüft werden, ob $\mathcal{M}, s \models \psi$ gilt.

```
function Sat (φ: Formula): set of State;
(* Vorbedingung: M = (S, →, l) *)
begin
  if φ ∈ AP ⟶ return {s | φ ∈ l(s)}
  [] φ = ¬φ1 ⟶ return S \ Sat(φ1)
  [] φ = φ1 ∨ φ2 ⟶ return Sat(φ1) ∪ Sat(φ2)
  [] φ = EX φ1 ⟶ return {s ∈ S | s → s' ∧ s' ∈ Sat(φ1)}
  [] φ = AF φ1 ⟶ return Sat_AF(φ1)
  [] φ = E[φ1 U φ2] ⟶ return Sat_EU(φ1, φ2)
  fi
end
(* Nachbedingung: Sat(φ) = {s ∈ S | M, s ⊨ φ} *)
```

Listing 4.5: Hauptprogramm des CTL Model Checking Algorithmus

Die eigentliche Berechnung von $Sat(\phi)$ erfolgt wie üblich durch Fallunterscheidung nach der konkreten Form von $\phi$. Handelt es sich um ein Propositionalzeichen $p \in \mathcal{AP}$, so enthält $Sat(p)$ alle Zustände, welche mit $p$ markiert sind, d.h. alle $s \in S$ für die $p \in l(s)$ gilt. Für die Negation $\neg\phi$ wird lediglich das Komplement von $Sat(\phi)$ (bzgl. $S$) berechnet. Die Disjunktion entspricht der Vereinigung.

Für $\mathbf{EX}\,\phi$ wird zunächst rekursiv die Menge $Sat(\phi)$ berechnet und anschliessend alle Zustände $s$, welche einen Zustand $s' \in Sat(\phi)$ in einem Schritt erreichen können, d.h. für die $s \to s'$ gilt. Formeln der Gestalt $\mathbf{AF}\,\phi$ und $\mathbf{E}[\phi\,\mathbf{U}\,\psi]$ werden durch die speziellen Funktionen $Sat_{AF}$ und $Sat_{EU}$ behandelt, welche in Listing 4.6 und Listing 4.7 dargestellt sind.

```
function Sat_AF (φ: Formula): set of State;
(* Vorbedingung: M = (S, →, l) *)
begin var S1, S2: set of State;
  S1, S2 := S, Sat(φ);
  do S1 ≠ S2 ⟶
    S1, S2 := S2, S2 ∪ {s ∈ S | ∀s' ∈ S : s → s' ⇒ s' ∈ S2}
  od;
  return S2
end
(* Nachbedingung: Sat_AF(φ) = {s ∈ S | M, s ⊨ AF φ} *)
```

Listing 4.6: Algorithmus für $Sat(\mathbf{AF}\,\phi)$

Im Fall von $\mathbf{AF}\,\phi$ ist die Idee denkbar einfach: Begonnen wird mit der Menge aller Zustände für die $\phi$ gilt (denn nach Definition der Semantik gilt in diesen Zuständen auch $\mathbf{AF}\,\phi$). Diese Zustandsmenge wird dann solange iterativ vergrössert, bis alle Zustände erfasst sind, für die $\mathbf{AF}\,\phi$ gilt. Dabei werden in jedem Schritt alle Zustände $s \in S$

hinzugenommen, von denen aus nur Zustände $s' \in S$ erreichbar sind, die sich bereits in der Zustandsmenge für $\mathbf{AF}\,\phi$ befinden.

```
function Sat_EU (φ, ψ: Formula): set of State;
(* Vorbedingung: M = (S, →, l) *)
begin var S1, S2, S3: set of State;
   S1, S2, S3 := Sat(φ), S, Sat(ψ);
   do S2 ≠ S3 ⟶
      S2, S3 := S3, S3 ∪ (S1 ∩ {s ∈ S | ∃s' ∈ S : s → s' ∧ s' ∈ S3})
   od;
   return S3
end
(* Nachbedingung: Sat_EU(φ, ψ) = {s ∈ S | M, s ⊨ E[φ U ψ]} *)
```

Listing 4.7: Algorithmus für $Sat(\mathbf{E}[\phi\,\mathbf{U}\,\psi])$

Der Algorithmus zur Berechnung von $Sat(\mathbf{E}[\phi\,\mathbf{U}\,\psi])$ in Listing 4.7 ist ähnlich einfach zu verstehen, wenn man sich ins Gedächtnis zurückruft, wie die Semantik der Formel definiert ist: $\mathcal{M}, s \models \mathbf{E}[\phi\,\mathbf{U}\,\psi]$ gilt genau dann, wenn ein Pfad $\pi \in \Pi_{\mathcal{M}}(s)$ und ein $i \geq 1$ existieren, so dass $\psi$ im Zustand $\pi(i)$ gilt und für alle vorangegangenen Zustände auf dem Pfad $\pi$ die Formel $\phi$ gilt. Entsprechend beginnt der Algorithmus mit der Menge aller Zustände, die $\psi$ erfüllen, denn diese erfüllen nach Definition auch $\mathbf{E}[\phi\,\mathbf{U}\,\psi]$. Diese Zustandsmenge wird dann wieder iterativ vergrössert, wobei in jedem Iterationsschritt alle Zustände $s \in S$ aufgenommen werden, die einerseits $\phi$ erfüllen und andererseits einen Nachfolger $s' \in S$ besitzen, der sich bereits in der Zustandsmenge befindet.

### Komplexitätsanalyse

Die Laufzeitkomplexität des Gesamtalgorithmus wird wie folgt bestimmt. Zunächst ist offensichtlich, dass die Berechnung von $Sat(\phi)$ zu Aufrufen von $Sat(\psi)$ für alle Teilformeln $\psi \in Sub(\phi)$ führt. Die Komplexität ist also proportional zu $|Sub(\phi)|$, und damit proportional zu $|\phi|$. Die Komplexität von $Sat_{AF}(\phi)$ ist proportional zu $|S| \cdot |\rightarrow|$, wobei $|S|$ die Anzahl der Zustände des Systems und $|\rightarrow|$ die Grösse der Übergangsrelation bezeichnet, denn die maximale Anzahl an Iteration ist $|S|$ – ausgehend von der Annahme, dass mit der leeren Menge begonnen wird, und in jeder Iteration nur ein einziger Zustand in die Menge $S_2$ aufgenommen wird – und für die Berechnung des neuen $S_2$ müssen jeweils alle Transitionen berücksichtigt werden. Die Grösse der Übergangsrelation ist beschränkt durch $\mathcal{O}(|S|^2)$, womit sich insgesamt eine *worst-case Laufzeitkomplexität* von

$$\mathcal{O}(|\phi| \cdot |S|^3)$$

ergibt. Die Laufzeit des Algorithmus ist also linear in der Länge der CTL-Formel, aber kubisch in der Grösse des Modells, was keinesfalls ein befriedigendes Ergebnis darstellt.

### Verbesserungen

Clarke, Emerson und Sistla zeigten 1986 in [CES86], dass sich diese Komplexität reduzieren lässt, indem eine effizientere Realisierung für Formeln der Gestalt $\mathbf{EG}\,\phi$ gewählt wird. Die Idee hierbei ist eine andere adäquate Menge von CTL-Formeln zu wählen, die über den temporalen Verknüpfungen **EX**, **EU** und **EG** gebildet wird. Die Menge $Sat_{EG}(\phi)$ wird dann durch den in Listing 4.8 dargestellten Algorithmus berechnet.

```
function Sat_EG (φ: Formula): set of State;
(* Vorbedingung: M = (S, →, l) *)
begin var S1, S2: set of State;
  S1, S2 := ∅, Sat(φ);
  do S1 ≠ S2 ⟶
    S1, S2 := S2, S2 ∩ {s ∈ S | ∃s' ∈ S : s → s' ∧ s' ∈ S2}
  od;
  return S2
end
(* Nachbedingung: Sat_EG(φ) = {s ∈ S | M, s ⊨ EG φ} *)
```

Listing 4.8: Algorithmus für $Sat(\mathbf{EG}\,\phi)$

Das Vorgehen ist dabei wie folgt: Zunächst werden alle Zustände berechnet, die $\phi$ erfüllen. Aus dieser Menge $S_2$ werden dann solange iterativ alle Zustände entfernt, die $\mathbf{EG}\,\phi$ nicht erfüllen, bis keine Veränderung mehr auftritt. Dabei wird in jedem Iterationsschritt nur diejenigen Zustände in $S_2$ belassen, die einen Nachfolger besitzen, der in $S_2$ enthalten ist. Unter Verwendung dieser Variante lässt sich die Laufzeitkomplexität des Model Checking Algorithmus für CTL auf

$$\mathcal{O}(|\phi| \cdot |S|^2)$$

reduzieren.

### Vergleich zum Algorithmus für LTL

In Abschnitt 4.4 wurde gezeigt, dass die Komplexität des LTL Model Checking Algorithmus exponentiell in der Länge der Formel ist. Die Differenz zur Komplexität des CTL Algorithmus scheint gewaltig, aber der Schein trügt an dieser Stelle. Beispielsweise kann man zeigen, dass für jedes Modell $\mathcal{M}$ eine LTL-Formel $\phi$ existiert, so dass jede zu $\mathbf{A}\,\phi$ oder $\mathbf{E}\,\phi$ äquivalente CTL-Formel – falls eine solche Formel denn existiert – exponentiell länger ist als $\phi$. Praktische Beispiele haben gezeigt, dass dieses Ergebnis keineswegs abwegig ist, und dass für realistische Spezifikationen LTL-Formeln tatsächlich um Grössenordnungen kürzer sind, als die entsprechenden CTL-Formeln.

Zusammenfassend lässt sich festhalten, dass für Eigenschaften, die sowohl in LTL als auch in CTL formuliert werden können, die kürzeste mögliche Formulierung in LTL niemals länger ist als die CTL-Formel, sondern sogar exponentiell kürzer sein kann. Dementsprechend hebt sich der augenscheinliche Vorteil, dass CTL Model Checking linear in der Länge der Formel ist, während LTL Model Checking exponentiell ist, angesichts der Tatsache auf, dass Eigenschaften in CTL eine (viel) längere Darstellung benötigen als in LTL.

### 4.6.2. Korrektheit

# A. Guarded Command Language

Die in Kapitel 4 vorgestellten Algorithmen werden in einer abstrakten Programmnotation angegeben, um unabhängig von einer speziellen Programmiersprache zu sein. Diese Notation basiert auf Dijkstra's Guarded Command Language [Dij75].

### Syntax

Ein Programm ist eine Folge von Anweisungen, die durch Semikolon getrennt werden, und ihrerseits aus Anweisungen bestehen können.

$$
\begin{array}{rcl}
\textit{statement} & ::= & \textbf{skip} \\
& | & \textbf{let } \textit{variable} \textbf{ in } \textit{set} \\
& | & \textit{variable} \ := \ \textit{expression} \\
& | & \textit{variable}, \textit{variable} \ := \ \textit{expression}, \textit{expression} \\
& | & \textbf{if } \textit{guarded command} \ [] \ldots [] \ \textit{guarded command} \textbf{ fi} \\
& | & \textbf{do } \textit{guarded command} \ [] \ldots [] \ \textit{guarded command} \textbf{ od} \\
& | & \textit{statement} \ ; \ \textit{statement} \\
\\
\textit{guarded command} & ::= & \textit{boolean expression} \longrightarrow \textit{statement}
\end{array}
$$

Kommentare werden wie in Pascal oder O'Caml von (* und *) umschlossen. Die Syntax von Ausdrücken und Bedingungen wird nicht näher festgelegt, stützt sich aber im wesentlichen auf die in der Mathematik benutzte Schreibweise.

### Semantik

Die Semantik entspricht größtenteils der intuitiven Bedeutung der Anweisungen im Kontext einer imperativen Programmiersprache, allerdings sind bedingte Anweisungen nicht deterministisch.

- **skip** steht für die leere Anweisung und bedeutet einfach „nichts tun".
- Die Anweisung **let** $x$ **in** $V$ bedeutet, dass auf nicht deterministische Weise ein beliebiges Element aus $V$ ausgewählt und an $x$ zugewiesen werden soll. Hierbei muss $V$ eine nicht leere Menge sein.

- Die einfache Zuweisung $x := e$ weist der Variablen $x$ den Wert des Ausdrucks $e$ zu.

- $x_1, x_2 := e_1, e_2$ steht für die *gleichzeitige* Zuweisung des Wertes von $e_1$ an $x_1$ und $e_2$ an $x_2$. Insbesondere kann $e_2$ die Variable $x_1$ enthalten und die Auswertung wird auf dem ursprünglichen Wert von $x_1$ durchgeführt.

- *Guarded commands* bestehen aus einem boolschen Ausdruck (dem sog. *guard*) und einer Anweisung. Die Anweisung darf nur dann ausgeführt werden, wenn der boolsche Ausdruck initial (siehe **if** und **do**) wahr ist. $(V \neq \emptyset) \longrightarrow x, y := 1, 2$ bedeutet zum Beispiel, dass die Zuweisung $x, y := 1, 2$ nur dann ausgeführt werden kann, wenn $V$ eine nicht leere Menge ist.

- **if** *guarded command*$_1$ [] ... [] *guarded command*$_n$ **fi** steht für eine bedingte Ausführung, bei der zunächst die boolschen Ausdrücke aller *guarded command*$_i$ mit $i \in \{1, \ldots, n\}$ geprüft werden und dann auf nicht deterministische Weise aus der Menge der wahren Ausdrücke ein *guarded command*$_j$ ausgewählt, dessen Anweisung dann ausgeführt wird. Existiert kein solches *guarded command*$_j$ wird die Ausführung der **if** Anweisung sofort beendet.

- **do** *guarded command*$_1$ [] ... [] *guarded command*$_n$ **od** steht für eine wiederholte Ausführung. Hierbei werden in jedem Durchlauf die boolschen Ausdrücke aller *guarded command*$_i$ mit $i \in \{1, \ldots, n\}$ geprüft und wie bei der **if** Anweisung auf nicht deterministische Weise die Anweisung eines *guarded command*$_j$ ausgeführt, dessen Bedingung wahr ist. Existiert kein solches *guarded command*$_j$ wird die wiederholte Ausführung sofort abgebrochen.

Um die Beschreibung des Algorithmus kurz zu halten, erlauben wir auch implizite Zuweisungen in den boolschen Ausdrücken der *guarded commands*[1]. Zum Beispiel ist es für eine LTL-Formel $\phi$ zulässig im Anweisungsteil $s$ von $\phi = \phi_1 \wedge \phi_2 \longrightarrow s$ die Variablen $\phi_1$ und $\phi_2$ zu benutzen.

### Datentypen

Als Datentypen wollen wir Mengen und Listen mit beliebigen Elementtypen zulassen. Für Mengen verwenden wir die übliche mathematische Notitation. Listen von Elementen schreiben wir als $\langle x_1, \ldots, x_n \rangle$, das Hinzufügen eines Elements $x$ zu einer Liste $L$ als $L^\frown\langle x \rangle$ bzw. $\langle x \rangle^\frown L$.

[1] Diese impliziten Zuweisungen entsprechen in etwa dem Pattern Matching Mechanismus von O'Caml und ähnlichen Programmiersprachen.

# Literaturverzeichnis


[Bro81] Brown, Douglas W.: A State-Machine Synthesizer – SMS. In: *DAC '81: Proceedings of the 18th conference on Design automation*. Piscataway, NJ, USA : IEEE Press, 1981, S. 301–305

[CE82] Clarke, Edmund M. ; Emerson, E. A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: *Logic of Programs, Workshop*. London, UK : Springer-Verlag, 1982. – ISBN 3–540–11212–X, S. 52–71

[CES86] Clarke, Edmund M. ; Emerson, E. A. ; Sistla, A. P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In: *ACM Transactions on Programming Languages and Systems* 8 (1986), Nr. 2, S. 244–263

[Cho74] Choueka, Yaacov: Theories of automata on $\omega$-tapes: a simplified approach. In: *J. Comput. System Sci.* 8 (1974), S. 117–141

[Dij75] Dijkstra, Edsger: Guarded commands, non-determinacy and formal derivation of programs. In: *Communications of the ACM* 18 (1975), August, Nr. 8, S. 453–457

[DP88] Dörfler, Willibald ; Peschek, Werner: *Einführung in die Mathematik für Informatiker*. München : Carl Hanser Verlag, 1988. – ISBN 3–446–15112–5

[EL85] Emerson, E. A. ; Lei, C. L.: Modalities for Model Checking: Branching Time Strikes Back. In: *ACM Symposium on Principles of Programming Languages*, 1985, S. 84–96

[GPVW95] Gerth, Rob ; Peled, Doron ; Vardi, Moshe ; Wolper, Pierre: Simple On-the-fly Automatic Verification of Linear Temporal Logic. In: *Protocol Specification Testing and Verification*. Warsaw, Poland : Chapman & Hall, 1995, S. 3–18

[HR04] Huth, Michael ; Ryan, Mark: *Logic in Computer Science: Modelling and Reasoning about Systems*. Second Edition. Cambridge University Press, 2004. – ISBN 0–521–54310–X

[Kat99] Katoen, Joost-Pieter: *Arbeitsberichte der Informatik*. Bd. 32-1: *Concepts, Algorithms, and Tools for Model Checking*. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999

[Kri63] Kripke, Saul A.: Semantical Considerations on Modal Logic. In: *Acta Philosophica Fennica* 16 (1963), S. 83–94

[Mar01] Martin, Alan: Adequate Sets of Temporal Connectives in CTL. In: *Electronic Notes Theoretical Computer Science* 52 (2001), Nr. 1

[Meu06] Meurer, Benedikt: PLTL Model Checking. In: *Seminar zur Theoretischen Informatik* (2006)

[Rud86] Rudell, Richard L.: Multiple-Valued Logic Minimization for PLA Synthesis / EECS Department, University of California, Berkeley. 1986 (UCB/ERL M86/65). – Forschungsbericht

[Spr91] Spreen, Dieter: *Logik für Informatiker*. 1991. – Vorlesungsskript

[Var95] Vardi, Moshe Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: *Banff Higher Order Workshop*, 1995, S. 238–266

[WVS83] Wolper, Pierre ; Vardi, Moshe Y. ; Sistla, A. P.: Reasoning about Infinite Computation Paths. In: *Proc. 24th IEEE Symposium on Foundations of Computer Science*. Tucson, 1983, S. 185–194