Implementierung eines Verfahrens zur Erzeugung von Büchi-Automaten aus LTL-Formeln in Isabelle

Alexander Schimpf
12. Dezember 2008

Diplomarbeit

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Arbeitsgruppe Grundlagen der Künstlichen Intelligenz

Erstgutachter: Prof. Dr. Bernhard Nebel
Zweitgutachter: Dr. Stephan Merz
Betreuer: Dr. Jan-Georg Smaus

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

In dieser Diplomarbeit wird mit Hilfe des interaktiven Theorembeweisers Isabelle/HOL ein Verfahren zur Erzeugung von Büchi-Automaten aus LTL-Formeln implementiert und die Korrektheit dieser Konstruktion bewiesen. Aus der Isabelle-Spezifikation wird schließlich unter Verwendung des Codegenerators von Isabelle/HOL ausführbarer Code der Implementierung erzeugt. Ausgehend von der ursprünglichen Idee des Verfahrens von Gerth, Peled, Vardi und Wolper konnte die Korrektheit der Konstruktion bestätigt werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Nach wie vor besteht eine große Herausforderung der Informatik darin, Systeme zu entwickeln, die zuverlässig funktionieren. Zu nahezu jedem Entwicklungsprozess eines Systems gehört eine Phase der Qualitätssicherung, die die Intention verfolgt, während der Entwicklung entstandene Fehler des Systems bereits in einem frühen Entwicklungsstadium zu entdecken. Klassische Qualitätssicherungsmaßnahmen beruhen auf den Prinzipien des Testens und Simulierens. Dabei wird im Wesentlichen das zugrunde liegende System auf repräsentativen Eingaben ausgeführt und das Ergebnis mit dem erwarteten Resultat verglichen. Ein ernsthaftes Problem dieser Verfahren besteht darin, dass in der Regel aus Zeit- und Kostengründen nur ein Bruchteil der möglichen Fälle abgedeckt werden kann, sodass eine klare Aussage darüber, ob und wie viele Fehler noch vorhanden sind bzw. vorhanden sein könnten, nicht gemacht werden kann. Ein alternatives Verfahren zur Qualitätssicherung des Entwicklungsprozesses wird als *formale Verifikation* bezeichnet. Dabei werden Eigenschaften des Systems formal nachgewiesen. Ein weit verbreiteter Ansatz, um einen Beweis für die Gültigkeit bestimmter Anforderungen eines Systems zu erbringen, heißt *Model Checking* [5] (*Modellprüfen*). Dabei handelt es sich um ein Verfahren, das das als zustandsbasierte Repräsentation vorliegende System bzgl. einer als temporallogische Formel[1] gegebenen Spezifikation automatisch überprüft.

Insbesondere bei größeren Eingaben stoßen klassische Modellprüfverfahren [4, 6] jedoch an ihre Grenzen und scheitern etwa an einer zu langen Ausführungszeit oder an einem zu hohen Speicherplatzverbrauch. Aus diesem Grund sind effizientere Verfahren erforderlich, die schneller eine Lösung finden und dabei weniger

[1] Siehe Abschnitt 2.2.

Speicherplatz verbrauchen. Problematisch ist nun jedoch die Tatsache, dass im Zuge der Entwicklung eines effizienten Verfahrens wiederum Fehler unterlaufen können, die ggf. dazu führen, dass dieses Verfahren ein falsches Ergebnis liefert. Ein Modellprüfverfahren beinhaltet in der Regel eine nicht-triviale Komponente, die die Spezifikation des Systems in eine zustandsbasierte Repräsentation überführt. Aufgrund der Komplexität dieser Komponente besteht die Gefahr, dass deren Implementierung Fehler enthält. Um diesen fatalen Fall ausschließen zu können, bietet es sich an, die kritischen Teile eines Modellprüfverfahrens selbst bzgl. deren Spezifikation zu überprüfen. Für diesen Zweck besteht die Möglichkeit einen weiteren Ansatz zu verwenden, um ein Verfahren gegen seine Spezifikation zu verifizieren: Mit Hilfe des interaktiven Theorembeweisers Isabelle/HOL implementiert man das Verfahren und prüft diese Implementierung dahingehend, ob sie sich bzgl. ihrer Spezifikation korrekt verhält. Der Codegenerator von Isabelle/HOL liefert schließlich aus der Isabelle-Implementierung ein Programm in einer funktionalen Programmiersprache. Im Folgenden werde ich meine Isabelle-Implementierung des Verfahrens von Gerth et al. [1] vorstellen, das ausgehend von einer Spezifikation, gegeben durch eine temporale Formel, einen so genannten Büchi-Automaten generiert. Dieses Verfahren stellt dabei einen wesentlichen Teil des Model-Checking-Prozesses dar.

Bevor ich dazu komme, die Isabelle-Implementierung vorzustellen, gehe ich kurz auf das Thema Model Checking in Kapitel 2 ein. Dabei befasse ich mich u. a. mit der linearen temporalen Logik und mit den Büchi-Automaten. Weiterhin gehe ich kurz auf den Model-Checking-Prozess ein, um zu verdeutlichen, welche Rolle das Verfahren von Gerth et al. [1] dabei spielt.

In Kapitel 3 skizziere ich zunächst die Idee dieses Verfahrens, auf dem meine Isabelle-Implementierung basiert, und gehe dabei auf den Vorschlag von Gerth et al. [1], das Verfahren zu implementieren, ein.

In Kapitel 4 stelle ich kurz den interaktiven Theorembeweiser Isabelle/HOL vor und erläutere dabei die in der Implementierung verwendeten Konzepte.

Anschließend stelle ich in Kapitel 5 meine Isabelle-Implementierung vor. Darin befasse ich mich sowohl mit der eigentlichen Isabelle-Spezifikation des Verfahrens als auch mit der Formalisierung und der Verifizierung der Korrektheitseigenschaften.

Schließlich befasse ich mich in Kapitel 6 mit dem Erzeugen von ausführbaren Code aus meiner Isabelle-Implementierung. Diese Implementierung wird anhand eines konkreten Beispiels demonstriert.

In Kapitel 7 schließe ich diese Ausarbeitung mit einem persönlichen Eindruck ab.

Kapitel 2

Model Checking

Beim Model Checking wird ein gegebenes System, etwa ein Programm oder aber eine logische Schaltung, bzgl. seiner Spezifikation automatisch überprüft. Dazu muss das System zunächst in eine geeignete Darstellung überführt werden. Eine Möglichkeit besteht darin, das System in eine zustandsbasierte Repräsentation abzubilden. Dabei repräsentiert ein Zustand etwa eine mögliche Konfiguration eines Programms oder eine potentielle Belegung der Eingänge einer logischen Schaltung. Zur Beschreibung der Korrektheit eines Systems nimmt man dabei Bezug auf elementare Eigenschaften, die über Zuständen ausgewertet werden, wie etwa „die Variable x hat den Wert a" oder „am Eingang $A1$ liegt ein Signal an". Zustandsübergänge modellieren den zeitlichen Verlauf des Systems. Im Sinne einer logischen Schaltung könnte ein Zustandsübergang etwa durch die Änderung eines Eingangssignals beschrieben werden. Eine geeignete Repräsentation für ein System ist die so genannte Kripke-Struktur.

2.1 Kripke-Struktur

Definition Eine *Kripke-Struktur* über einer endlichen Menge Prop von atomaren Aussagen [2] ist ein Tupel (S, S_0, δ, L). Hierbei ist:

- S eine endliche Menge von Zuständen;
- S_0 eine Menge von Startzuständen;
- $\delta \subseteq S \times S$ eine totale Transitionsrelation, d. h. für alle $s \in S$ existiert ein $s' \in S$ mit der Eigenschaft $(s, s') \in \delta$;

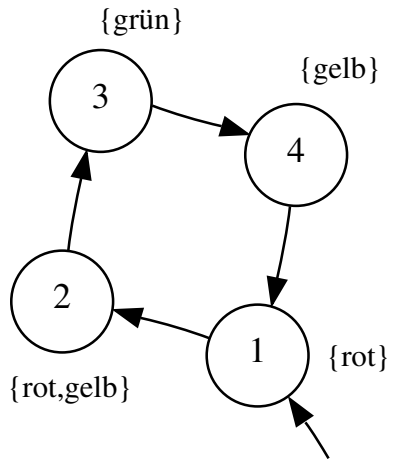

Abbildung 2.1: Ampelschaltung als Kripke-Struktur

- $L : S \to 2^{\mathsf{Prop}}$ eine Markierungsfunktion.

Die Zustände einer Kripke-Struktur (S, S_0, δ, L), also die Elemente von S, repräsentieren Zustände des Systems, während die Transitionen aus δ den zeitlichen Verlauf des Systems wiedergeben. Startzustände, also Elemente von S_0, sind Zustände des Systems, bei denen die Ausführung des System beginnen könnte. Die Elemente von Prop sind Eigenschaften des Systems, die in einem Zustand erfüllt oder nicht erfüllt sein können. Die Markierungsfunktion L ordnet jedem Zustand eine Menge von Propositionen zu, die in diesem Zustand erfüllt sind.

Beispiel Abbildung 2.1 zeigt eine Kripke-Struktur für eine einfache Ampelschaltung als Diagramm. Die Kreise repräsentieren die Zustände dieser Schaltung, während die Pfeile die Zustandsübergänge modellieren. Der Zustand mit der Nummer 1, der eine eingehende Kante besitzt, die von keinem anderen Zustand ausgeht, wird als Startzustand betrachtet. Die Menge Prop ist gegeben durch $\{\text{rot}, \text{gelb}, \text{grün}\}$. Die Mengen neben den jeweiligen Zuständen entsprechen dem Wert der Markierungsfunktion.

Definition Ein *Pfad* σ einer Kripke-Struktur (S, S_0, δ, L) ist eine unendliche Folge von Zuständen $\sigma = s_0 s_1 s_2 ...$ mit $s_0 \in S_0$ und $(s_i, s_{i+1}) \in \delta$ für $i \in \mathbb{N}$. Man bezeichnet unendliche Folgen $a_0 a_1 a_2 ...$ über einem Alphabet A (d. h. $a_i \in A$ für $i \in \mathbb{N}$) auch als ω-Wörter über A. Folglich ist ein Pfad σ einer Kripke-Struktur

(S, S_0, δ, L) ein ω-Wort über S. Die i-te Komponente eines ω-Wortes w liefert $w(i)$. Das k-Suffix eines ω-Wortes $w = a_0a_1a_2...$ (kurz: w_k) ist gegeben durch $a_ka_{k+1}a_{k+2}...$.

Ein Pfad σ einer Kripke-Struktur (S, S_0, δ, L), die ein System modelliert, repräsentiert eine mögliche zeitliche Abfolge von Zuständen. Jeder Pfad σ besitzt eine unendliche Abfolge von Eigenschaften gegeben durch $L(\sigma(0))L(\sigma(1))L(\sigma(2))...$ (kurz: $L(\sigma)$). Diese kann als ω-Wort über 2^{Prop} aufgefasst werden.

Beispiel Ausgehend von der Kripke-Struktur aus Abbildung 2.1 erfüllt die unendliche Folge 1 2 3 4 1 2 3 4 ... alle Anforderungen eines Pfades. Durch die unendliche Folge {rot} {rot, gelb} {grün}{gelb} {rot} {rot, gelb} {grün}{gelb}... hat man die dazugehörige unendliche Abfolge von Eigenschaften.

Definition Die *Sprache* einer Kripke-Struktur $\mathcal{K} = (S, S_0, \delta, L)$ ist gegeben durch $lang(\mathcal{K}) := \{L(\sigma) \mid \sigma \text{ ist Pfad von } \mathcal{K}\}$.

Ein Element von $lang(\mathcal{K})$ ist eine unendliche Folge von Eigenschaften, die das zu $\mathcal{K}$ dazugehörige System erfüllt.

Beispiel Die Sprache der Kripke-Struktur aus Abbildung 2.1 ist gegeben durch die Menge {{rot} {rot, gelb} {grün} {gelb} {rot} {rot, gelb} {grün} {gelb}...}, die also aus genau einem ω-Wort über 2^{Prop} besteht[1].

Nun stellt sich die Frage, ob Folgen von Eigenschaften auch bestimmte zustandsübergreifende Bedingungen erfüllen. Um solche zustandsübergreifende Spezifikationen formulieren zu können, verwendet man die so genannte lineare temporale Logik.

2.2 Lineare temporale Logik (LTL)

Eine lineare temporale Logik [1, Kap. 2] ist gegeben durch die Mengen Prop und Φ, die logischen Symbole $\neg, \vee, \mathsf{X}$ und U und die Relation $\models$. Hierbei ist:

- Prop eine endliche, nicht leere Menge von Propositionen;

[1] Da Systeme in der Regel nichtdeterministisch sind, existieren im Allgemeinen sehr viele dazugehörige Eigenschaftsfolgen.

- Φ die Menge von LTL-Formeln;
- $\neg$ und $\vee$ die Booleschen Operatoren für Negation und Disjunktion;
- X und U die temporalen Operatoren Next und Until;
- $\vDash$ eine Relation, die ein ω-Wort über 2^{Prop} und eine LTL-Formel in Beziehung setzt.

Die Menge Φ der LTL-Formeln ist wie folgt induktiv definiert:

- $\mathsf{Prop} \subseteq \Phi$;
- wenn $\phi \in \Phi$ und $\psi \in \Phi$,
  dann gilt $\neg\phi \in \Phi$, $\phi \vee \psi \in \Phi$, $\mathsf{X}\phi \in \Phi$ und $\phi\mathsf{U}\psi \in \Phi$.

Folglich ist jede Proposition und jeder Ausdruck, der durch Kombination der Booleschen und temporalen Operatoren mit LTL-Formeln hervorgeht, eine LTL-Formel.

Die Semantik von LTL-Formeln wird für ein ω-Wort ξ über dem Alphabet 2^{Prop} definiert. Ein solches ω-Wort entspricht einer bestimmten Abfolge von propositionalen Interpretationen. Intuitiv formuliert, besagt es, welche Propositionen im Zeitverlauf gelten und welche nicht. Daher wird die Semantik von LTL-Formeln bzgl. eines solchen ω-Wortes definiert, d. h. die Semantik ist eine Relation die besagt, wann ein ω-Wort eine LTL-Formel erfüllt.

- $\xi \vDash p$ gdw. $p \in \xi(0)$ für $p \in \mathsf{Prop}$;
- $\xi \vDash \neg\phi$ gdw. $\xi \vDash \phi$ nicht gilt;
- $\xi \vDash \phi \vee \psi$ gdw. $\xi \vDash \phi$ gilt oder $\xi \vDash \psi$ gilt;
- $\xi \vDash \mathsf{X}\phi$ gdw. $\xi_1 \vDash \phi$ gilt;
- $\xi \vDash \phi\mathsf{U}\psi$ gdw. es existiert ein $i \in \mathbb{N}$ mit $\xi_i \vDash \psi$ und für alle $j \in \mathbb{N}$ mit $j < i$ gilt $\xi_j \vDash \phi$.

Das ω-Wort ξ bezeichnet man als *Interpretation*. Für $\xi \vDash \phi$ sagt man auch: ξ ist ein Modell für ϕ.

Für einige spezielle LTL-Formeln existieren Abkürzungen, und zwar wie folgt:

- True: $\top := p \vee \neg p$ für $p \in \mathsf{Prop}$;
- False: $\bot := \neg\top$;
- Konjunktion: $\phi \wedge \psi := \neg(\neg\phi \vee \neg\psi)$;
- Implikation: $\phi \rightarrow \psi := \neg\phi \vee \psi$;
- Release: $\phi \mathsf{V} \psi := \neg(\neg\phi \mathsf{U} \neg\psi)$;
- Finally: $\mathsf{F}\phi := \top \mathsf{U} \phi$;
- Globally: $\mathsf{G}\phi := \neg\mathsf{F}\neg\phi$.

Die so definierte temporale Logik LTL wird als Beschreibungssprache verwendet, um Spezifikationen für eine Kripke-Struktur anzugeben. Die temporalen Operatoren erlauben es, Aussagen zustandsübergreifend zu formulieren. Mit $\mathsf{X}\phi$ lässt sich die Forderung ausdrücken, dass ϕ im nächsten Zustand erfüllt sein muss, während $\phi \mathsf{U} \psi$ fordert, dass ψ in irgendeinem Zustand eines Pfades erfüllt sein muss und dass in allen Zuständen auf dem Pfad dorthin stets ϕ gilt. Analog dazu fordert $\mathsf{F}\phi$, dass ϕ in irgendeinem Zustand eines Pfades erfüllt ist und $\mathsf{G}\phi$ bedingt, dass ϕ stets erfüllt ist. Aufgrund der Äquivalenz $\xi \vDash \phi \mathsf{U} \psi \equiv \xi \vDash \neg(\neg\phi \mathsf{V} \neg\psi)$ wird der V-Operator auch als der zu U duale Operator bezeichnet. Analog dazu ist der Operator $\wedge$ dual zum Operator $\vee$. Die Semantik der Formel $\phi \mathsf{V} \psi$ ist so zu verstehen, dass wenn ϕ für einen Zustand erfüllt ist, dann die Gültigkeit von ψ für alle Folgezustände nicht mehr erforderlich ist. Ist jedoch ϕ für keinen der Vorgängerzustände eines beliebigen Zustandes erfüllt, so muss unmittelbar ψ gelten.

Beispiel Bezogen auf die Kripke-Struktur aus Abbildung 2.1 könnte man sich fragen, ob diese die LTL-Formel $\mathsf{G}(\text{rot} \rightarrow \mathsf{F}\,\text{grün})$ erfüllt, d. h. ob von jedem Zustand aus, in dem rot erfüllt ist, ein Zustand erreichbar ist, in dem grün erfüllt ist.

2.3 Verifikation der Spezifikation

Sei nun die Spezifikation als LTL-Formel ϕ und das System als Kripke-Struktur $\mathcal{K}$ gegeben. Es gilt: $\mathcal{K}$ erfüllt ϕ, falls die Teilmengenbeziehung $lang(\mathcal{K}) \subseteq \{\xi \mid \xi \vDash \phi\}$ zutrifft, d. h. falls für alle Pfade σ von $\mathcal{K}$ die dazugehörige Abfolge von Eigenschaften $L_{\mathcal{K}}(\sigma)$[2] die LTL-Formel ϕ erfüllt. Gibt es nun einen Pfad σ von $\mathcal{K}$, für

[2] Die Funktion $L_{\mathcal{K}}$ ist die Markierungsfunktion von $\mathcal{K}$.

den gilt, dass $L_{\mathcal{K}}(\sigma)$ die Spezifikation ϕ verletzt, d. h. es gilt nicht $L_{\mathcal{K}}(\sigma) \vDash \phi$ bzw. es gilt gemäß der Definition der Semantik von LTL-Formeln aus Abschnitt 2.2 $L_{\mathcal{K}}(\sigma) \vDash \neg\phi$, dann hat man die Situation gegeben, in der folgende Eigenschaft erfüllt ist:

$$lang(\mathcal{K}) \cap \{\xi \mid \xi \vDash \neg\phi\} \neq \emptyset$$

Folglich kann man, um die Erfüllbarkeit von ϕ durch $\mathcal{K}$ zu zeigen, auch die Eigenschaft nachweisen, die besagt, dass es keinen Pfad σ von $\mathcal{K}$ gibt, sodass die Spezifikation ϕ durch $L_{\mathcal{K}}(\sigma)$ verletzt wird, also:

$$lang(\mathcal{K}) \cap \{\xi \mid \xi \vDash \neg\phi\} = \emptyset$$

Diese Tatsache wird von Modellprüfverfahren aufgegriffen, um die Spezifikation bzgl. des Systems automatisch zu überprüfen. Dazu wird zunächst sowohl das System als auch die negierte Spezifikation in einen so genanten ω-Automaten, hier ein „markierter verallgemeinerter Büchi"-Automat, überführt, um dann mittels graphentheoretischer Methoden [8] zu verifizieren, dass der Produktautomat[3] der beiden Automaten gerade die leere Sprache erkennt. Während eine Kripke-Struktur direkt in einen ω-Automat überführt werden kann (vgl. Abschnitt 2.3.2), haben Vardi und Wolper gezeigt [6], dass zu jeder LTL-Formel ϕ ein ω-Automat konstruiert werden kann, der exakt die ω-Wörter akzeptiert, die ϕ erfüllt.

2.3.1 Markierter verallgemeinerter Büchi-Automat (labeled generalized Büchi automaton, LGBA)

Die Definition des LGBA [1, Kap. 3] basiert auf dem so genannten verallgemeinerten Büchi-Automaten (generalized Büchi automaton, GBA).

Definition Ein GBA $\mathcal{A}$ ist ein Tupel (Q, I, δ, F), sodass gilt:

- Q ist eine endliche Menge von Zuständen;
- $I \subseteq Q$ ist die Menge der Startzuständen;
- $\delta \subseteq Q \times Q$ ist die Transitionsrelation;

[3] Ein Produktautomat für zwei Automaten A_1 und A_2 akzeptiert nur die Wörter, die sowohl von A_1 als auch von A_2 akzeptiert werden.

- $F \subseteq 2^{2^Q}$ ist die Menge von Mengen akzeptierender Zustände.

Ein ω-Wort σ über Q heißt *Pfad* von $\mathcal{A}$, falls $\sigma(0) \in I$ und $(\sigma(i), \sigma(i+1)) \in \delta$ für $i \in \mathbb{N}$. Das Limit für ein ω-Wort ξ ist gegeben durch[4]:

$$\mathsf{limit}(\xi) := \{a | \exists_\infty n \in \mathbb{N}. \xi(n) = a\}$$

Der GBA $\mathcal{A}$ *akzeptiert* einen Pfad σ von $\mathcal{A}$, falls für alle $M \in F$ die Bedingung $\mathsf{limit}(\sigma) \cap M \neq \emptyset$ erfüllt ist.

Basierend auf der Definition des GBA definiert man nun den LGBA.

Definition Ein LGBA ist ein Tripel $(\mathcal{A}, \mathcal{D}, \mathcal{L})$, sodass gilt:

- $\mathcal{A} = (Q, I, \delta, F)$ ist ein GBA;
- $\mathcal{D}$ ist eine endliche Menge von Markierungen;
- $\mathcal{L} : Q \to 2^{\mathcal{D}}$ ist die Markierungsfunktion.

Ein Pfad σ von $\mathcal{A}$ akzeptiert ein ω-Wort ξ über $\mathcal{D}$, falls $\xi(i) \in \mathcal{L}(\sigma(i))$ für $i \in \mathbb{N}$ erfüllt ist. Ein LGBA akzeptiert ein ω-Wort ξ über $\mathcal{D}$ gdw. ein Pfad von $\mathcal{A}$ existiert, der ξ akzeptiert und der von $\mathcal{A}$ akzeptiert wird.

2.3.2 LGBA aus einer Kripke-Struktur

Sei $\mathcal{K} = (S, S_0, \delta, L)$ eine Kripke-Struktur über Prop. Der LGBA $\mathfrak{A} = (\mathcal{A}, \mathcal{D}, \mathcal{L})$, der exakt die ω-Wörter akzeptiert, die in $lang(\mathcal{K})$ enthalten sind, kann wie folgt konstruiert werden:

- $\mathcal{A} = (S, S_0, \delta, \emptyset)$ der GBA von $\mathfrak{A}$;
- $\mathcal{D} = 2^{\mathsf{Prop}}$;
- $\mathcal{L} : S \to 2^{2^{\mathsf{Prop}}}$ gegeben durch $\mathcal{L}(s) := \{L(s)\}$.

[4] Das Symbol $\exists_\infty$ bedeutet es existieren unendlich viele.

Da die Menge der Akzeptanzmengen des GBA $\mathcal{A}$ gerade leer ist, akzeptiert der Automat alle Pfade von $\mathcal{A}$. Dabei ist ein Pfad des GBA $\mathcal{A}$ identisch mit dem Pfad der Kripke-Struktur $\mathcal{K}$. Für einen Pfad σ und $i \in \mathbb{N}$ hat man $\mathcal{L}(\sigma(i)) = \{L(\sigma(i))\}$. Die Akzeptanzbedingung $\xi(i) \in \mathcal{L}(\sigma(i))$ bzw. $\xi(i) \in \{L(\sigma(i))\}$ liefert schließlich die Bedingung, dass nur solche ω-Wörter ξ akzeptiert werden, die $\xi(i) = L(\sigma(i))$ für $i \in \mathbb{N}$ bzw. $\xi = L(\sigma)$ erfüllen. Folglich akzeptiert der LGBA $\mathfrak{A}$ nur ω-Wörter der Form $L(\sigma)$, da für jedes ω-Wort dieser Art ein Pfad von $\mathcal{A}$ existiert, der es akzeptiert.

Mit der Konstruktion eines LGBA aus einer LTL-Formel beschäftigt sich das nächste Kapitel 3.

Kapitel 3

LGBA-Konstruktion aus einer LTL-Formel

Sei eine LTL-Formel ϕ gegeben. Im Folgenden stelle ich kurz die Idee des Verfahrens von Gerth et al. [1] zur Erzeugung des LGBA $\mathfrak{A}_\phi$, der exakt die Modelle über 2^{Prop} für ϕ akzeptiert, vor. Anders ausgedrückt, erfüllt der konstruierte LGBA $\mathfrak{A}_\phi$ folgende Eigenschaft: $\xi \vDash \phi$ gdw. $\mathfrak{A}_\phi$ akzeptiert ξ, für ein ω-Wort ξ über 2^{Prop}. Bereits aus dieser Eigenschaft wird ersichtlich, dass die Menge von Markierungen des LGBA $\mathfrak{A}_\phi$ gerade als 2^{Prop} gewählt werden muss.

Das besagte Verfahren erzeugt zunächst nur einen Graphen, aus dem jedoch dann der entsprechende LGBA $\mathfrak{A}_\phi$ konstruiert wird. Dieser Graph enthält bereits die Zustände und die Zustandsübergänge des LGBA $\mathfrak{A}_\phi$. Um den LGBA zu vervollständigen, benötigt man außerdem noch die Menge der Startzustände, die Menge der Akzeptanzmengen und die Markierungsfunktion.

Definition Ein *Graph* ist ein Paar (V, Δ). Dabei gilt:

- V eine Menge von Zuständen;
- $\Delta : V \to 2^V$ eine Transitionsfunktion.

Für zwei Zustände q und q' sagt man, dass eine Kante von q nach q' existiert, falls $q \in \Delta(q')$ erfüllt ist.

Ein Graph wird üblicherweise durch eine Menge von Zuständen und durch eine Menge von Kanten definiert. In Hinsicht auf das hier vorgestellte Verfahren, ist

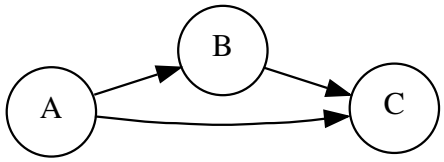

Abbildung 3.1: Beispiel eines Graphen

es jedoch von Vorteil, statt der Menge von Kanten eine Transitionsfunktion zu verwenden. Der Bezug zwischen dieser Definition und dem Verfahren wird in Abschnitt 3.1.2 erläutert.

Beispiel In Abbildung 3.1 wird ein Graph durch ein Diagramm dargestellt. Die Menge der Zustände V ist gegeben durch $\{A, B, C\}$. Die Transitionsfunktion ist durch $\Delta(A) = \{\}$, $\Delta(B) = \{A\}$ und $\Delta(C) = \{A, B\}$ definiert.

Das hier vorgestellte Verfahren lässt sich in zwei Teile gliedern: Konstruktion des Graphen und Transformation des Graphen in einen LGBA.

3.1 Konstruktion des Graphen

Die Konstruktion des Graphen erfordert zunächst, dass die Eingabeformel in Negationsnormalform vorliegt.

3.1.1 Negationsnormalform

Definition Eine LTL-Formel ϕ ist in *Negationsnormalform*, falls das Negationszeichen $\neg$ in ϕ höchstens direkt vor einer Proposition vorkommt.

Jede LTL-Formel kann unter Verwendung der in Abschnitt 2.2 definierten Abkürzungen in eine semantisch identische Formel in Negationsnormalform überführt werden. Dazu führt man solange Umformungen durch, bis die entsprechende Form vorliegt. Dabei verschiebt sich das Negationszeichen sukzessive von außen nach innen. In der Tat wurde der zum Until-Operator duale Release-Operator V gerade für diese Zwecke eingeführt. Die dazu erforderlichen Umformungsregeln sind die folgenden:

- $\xi \vDash \neg\top \equiv \xi \vDash \bot$
- $\xi \vDash \neg\bot \equiv \xi \vDash \top$
- $\xi \vDash \neg\neg\phi \equiv \xi \vDash \phi$
- $\xi \vDash \neg(\phi \vee \psi) \equiv \xi \vDash \neg\phi \wedge \neg\psi$
- $\xi \vDash \neg(\phi \wedge \psi) \equiv \xi \vDash \neg\phi \vee \neg\psi$
- $\xi \vDash \neg(\phi \rightarrow \psi) \equiv \xi \vDash \phi \wedge \neg\psi$
- $\xi \vDash \neg(\mathsf{X}\phi) \equiv \xi \vDash \mathsf{X}(\neg\phi)$
- $\xi \vDash \neg(\phi \mathsf{U} \psi) \equiv \xi \vDash \neg\phi \mathsf{V} \neg\psi$
- $\xi \vDash \neg(\phi \mathsf{V} \psi) \equiv \xi \vDash \neg\phi \mathsf{U} \neg\psi$
- $\xi \vDash \neg\mathsf{F}\phi \equiv \xi \vDash \bot \mathsf{V} \neg\phi$
- $\xi \vDash \neg\mathsf{G}\phi \equiv \xi \vDash \mathsf{F}\neg\phi$

Die Idee, die hinter der Negationsnormalform steckt, ist, dass man durch die Transformation in Negationsnormalform statt dem allgemeinen Negationsfall $\neg\psi$ den speziellen Fall $\neg p$ für $p \in \mathsf{Prop}$ erhält. In der Regel ist dieser spezielle Fall einfacher zu handhaben, insbesondere dann, wenn man eine Funktion über der Struktur der LTL-Formeln definiert.

3.1.2 Idee der Graphenkonstruktion

Sei nun also die Eingabeformel in Negationsnormalform. Die Konstruktion des Graphen basiert hauptsächlich auf der Idee, eine LTL-Formel bzgl. ihrer Struktur so zerlegen zu können, dass sich ein Teil der Aussage auf den aktuellen, ein anderer Teil auf den nächsten Zustand bezieht. Sei etwa die Eingabeformel ϕ gegeben durch $\phi = \neg p \wedge \mathsf{X}\psi$. Man erhält durch $\neg p$ den Teil, der unmittelbar gilt, und durch ψ den Teil, der im nächsten Zustand erfüllt sein muss. Die wichtigste Umformung dieser *Expansionsstrategie* ist jedoch die, die sich auf den U-Operator bezieht: $\xi \vDash \mu \mathsf{U} \psi \equiv \xi \vDash \psi \vee (\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi))$[1].

Mit dieser Expansionsstrategie erzeugt das Verfahren die Zustände und die Transitionen des Graphen sukzessive. Dabei wird zunächst der Teil der Eingabeformel

[1] Die Beweisidee für diese Äquivalenz findet man in Abschnitt 5.1.2.

extrahiert und in geeigneter Weise zu einem Zustand des Graphen verarbeitet, der unmittelbar erfüllt sein muss. Gleichzeitig wird der Teil, der im nächsten Zustand erfüllt sein muss, vorgemerkt, um diesen dann im Folgezustand wieder auf die gleiche Art und Weise analysieren und verarbeiten zu können. Im Wesentlichen aufgrund der Disjunktion kann es dabei beim Verarbeiten der Eingabeformel zum Nichtdeterminismus kommen, d. h. es ist möglich, dass ein Zustand des Graphen mehrere ausgehende Kanten besitzt.

Die Abbildung 3.3 auf Seite 16 zeigt den Algorithmus von Gerth et al. [1, Fig. 1], der den Graphen konstruiert, in Pseudo-Code. Dieser Algorithmus basiert auf einer Datenstruktur (Abbildung 3.3, Zeile 1-2), die für die Repräsentation von Automatenzuständen verwendet wird. Diese Zustandsrepräsentation besitzt folgende Felder:

- *Name:* Ein String, der den Namen des Zustands darstellt.
- *Father*: Ein String, der dazu verwendet wird, Zustände mit anderen Zuständen anhand des Zustandsnamen in eine Beziehung zu bringen. Dieses Feld ist nur für den Korrektheitsbeweis von Bedeutung.
- *Incoming:* Eine Menge von Zustandsnamen, von deren Zuständen eine ausgehende Kante, die im aktuellen Zustand eingeht, existiert. Ist der spezielle Bezeichner init in dieser Menge enthalten, so gilt der aktuelle Zustand als Startzustand.
- *New*: Eine Menge von LTL-Formeln, die im aktuellen Zustand erfüllt sein müssen und die vom Algorithmus noch nicht verarbeitet wurden.
- *Old*: Eine Menge von LTL-Formeln, die im aktuellen Zustand erfüllt sein müssen und die vom Algorithmus bereits verarbeitet wurden.
- *Next*: Eine Menge von LTL-Formeln, die im nächsten Zustand erfüllt sein müssen.

Der Algorithmus selbst ist durch zwei Funktionen gegeben:

- **expand** (Abbildung 3.3, Zeile 3-32): Eine rekursive Funktion, die die Expansionsstrategie, wie oben beschrieben, implementiert.
- **create_graph** (Abbildung 3.3, Zeile 33-36): Eine Funktion, die die Funktion **expand** mit speziellen Werten aufruft. Die Ausgabe dieser Funktion

besteht aus einer Liste von Zuständen. Im Wesentlichen wird durch das *Incoming*-Feld dieser Zustände eine Transitionsfunktion impliziert (vgl. Abschnitt 5.2.3.5), sodass die Ausgabe im Sinne der Definition eines Graphen zu Beginn dieses Kapitels betrachtet werden kann.

Von der Funktion **expand** werden Hilfsfunktionen [1, Seite 7], welche in Abbildung 3.2 zu sehen sind, verwendet.

η	**New1**(η)	**Next1**(η)	**New2**(η)
$\mu \mathsf{U} \psi$	$\{\mu\}$	$\{\mu \mathsf{U} \psi\}$	$\{\psi\}$
$\mu \mathsf{V} \psi$	$\{\psi\}$	$\{\mu \mathsf{V} \psi\}$	$\{\mu, \psi\}$
$\mu \vee \psi$	$\{\mu\}$	$\emptyset$	$\{\psi\}$

Abbildung 3.2: Hilfsfunktionen nach Gerth et al. [1, Kap. 3.2]

Diese Hilfsfunktion extrahieren aus LTL-Formeln der Form $\mu \mathsf{U} \psi$, $\mu \mathsf{V} \psi$ und $\mu \vee \psi$ die notwendigen Informationen, um eine Verzweigung des Graphen einzuleiten. Für die Until-Formel der Form $\mu \mathsf{U} \psi$ erhält man gemäß der Äquivalenz $\xi \vDash \mu \mathsf{U} \psi \equiv \xi \vDash \psi \vee (\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi))$ durch **New2** den linken Teil der Disjunktion, durch **New1** den Teil des rechten Disjunktionsglieds, der unmittelbar erfüllt sein muss, und durch **Next1** den Teil des rechten Disjunktionsglieds, der im nächsten Zustand erfüllt sein muss.

Die Konstruktion des Graphen beginnt mit der Funktion **create_graph**, der als Parameter eine LTL-Formel in Negationsnormalform übergeben wird und die die Funktion **expand** mit einem Startzustand und einer leeren Menge von Zuständen aufruft. Letztere repräsentiert den Ausgabegraphen und wird im Zuge der Berechnung sukzessive erweitert. Die Funktion **expand** vervollständigt dann den aktuellen Zustand, der hier als Parameter der Funktion vorliegt, ausgehend von dessen *New*-Feld gemäß der oben beschriebenen Expansionsstrategie schrittweise. Ist die Konstruktion des aktuellen Zustands abgeschlossen, d. h. wurden alle LTL-Formeln, die im aktuellen Zustand erfüllt sein müssen, vom Algorithmus verarbeitet, so führt der Algorithmus die Konstruktion des Graphen (unter bestimmten Bedingungen) mit dem nächsten Zustand fort. Dabei analysiert der Algorithmus nach dem gleichen Schema die LTL-Formeln, die für den nächsten Zustand vorgemerkt wurden.

Wie im Einzelnen der Algorithmus funktioniert, wird in Kapitel 5 anhand meiner Implementierung detailliert erläutert.

```
1  record graph_node = [Name:string, Father:string, Incoming:set of string,
2        New:set of formula, Old:set of formula, Next:set of formula];

3  function expand (Node, Nodes_Set)
4     if New(Node)=∅ then
5        if ∃ND ∈Nodes_Set with Old(ND)=Old(Node) and Next(ND)=Next(Node)
6        then Incoming(ND) = Incoming(ND)∪Incoming(Node);
7           return(Nodes_Set);
8        else return(expand([Name⇐Father ⇐new_name(),
9           Incoming⇐ {Name(Node)}, New⇐Next(Node),
10          Old⇐ ∅, Next⇐ ∅], {Node}∪Nodes_Set))
11    else
12       let η ∈New;
13       New(Node) := New(Node)\{η};
14       case η of
15       η = P_n, or ¬P_n or η =T or η =F=>
16          if η =F or Neg(η) ∈Old(Node) (* Current node contains a contradiction *)
17             then return(Nodes_Set) (* Discard current node *)
18             else Old(Node):=Old(Node)∪{η};
19             return(expand(Node, Nodes_Set));
20       η = μ U ψ, or μVψ, or μ ∨ ψ =>
21          Node1:=[Name⇐new_name(), Father⇐ Name(Node), Incoming⇐Incoming(Node),
22             New⇐New(Node)∪({New1(η)}\Old(Node)),
23             Old⇐Old(Node)∪{η}, Next=Next(Node)∪{Next1(η)} ];
24          Node2:=[Name⇐new_name(), Father⇐Name(Node), Incoming⇐Incoming(Node),
25             New⇐New(Node)∪({New2(η)}\Old(Node)),
26             Old⇐Old(Node)∪{η}, Next⇐Next(Node)];
27          return(expand(Node2, expand(Node1, Nodes_Set)));
28       η = μ ∧ ψ =>
29          return(expand([Name⇐ Name(Node), Father⇐Father(Node), Incoming⇐Incoming(Node),
30             New⇐New(Node)∪({μ, ψ}\Old(Node)),
31             Old⇐Old(Node)∪{η}, Next=Next(Node)], Nodes_Set))
32end expand;

33function create_graph (φ)
34   return(expand([Name⇐Father⇐new_name(), Incoming⇐{init},
35      New⇐ {φ}, Old⇐ ∅, Next⇐ ∅], ∅))
36end create_graph;
```

Abbildung 3.3: Der Algorithmus von Gerth et al. [1, Fig. 1] zur Konstruktion des Graphen aus einer LTL-Formel

3.2 Transformation des Graphen in einen LGBA

Wie bereits zu Beginn dieses Kapitels erwähnt, wird der LGBA aus dem konstruierten Graphen gewonnen. Dabei sind die Zustände und die Transitionen des Graphen gleichzeitig auch die Zustände und die Transitionen des LGBA. Ein Startzustand ist ein Zustand, von dem aus die Verarbeitung der Eingabeformel initiiert wurde, d. h. falls init in dessen *Incoming*-Feld enthalten ist. Wie bereits festgestellt, wählt man als Menge der Markierungen gerade 2^{Prop}. Folglich ordnet die Markierungsfunktion einem Zustand eine Menge von Mengen von Propositionen zu, und zwar gerade solche Mengen, die nur Propositionen enthalten, die in jenem Zustand erfüllt sein müssen und zu keinem Widerspruch[2] führen. Diese werden, wie bereits beschrieben, durch die Dekomposition der Eingabeformel ermittelt.

In der Tat lässt die Konstruktion des Graphen aus Abschnitt 3.1 für eine Until-Formel $\mu \mathsf{U} \psi$ unendliche Pfade zu, auf denen nie ein Zustand erreicht wird, in dem ψ erfüllt ist. Solche Pfade gilt es zu unterbinden, da sie der Semantik der Until-Formel widersprechen und somit nicht die Modelle dieser LTL-Formel implizieren. Dazu wird die Akzeptanzbedingung des LGBA so gewählt, dass unzulässige Pfade für Until-Formeln nicht akzeptiert werden.

Wie im einzelnen der LGBA aussieht, wie etwa die Akzeptanzmengen zu wählen sind oder wie die Markierungsfunktion tatsächlich beschaffen ist, wird in Abschnitt 5.3 ausführlich erläutert.

[2] Man erhält einen Widerspruch, wenn in einem Zustand sowohl p als auch $\neg p$ zu erfüllen sind.

Kapitel 4

Isabelle/HOL

Die Implementierung des Verfahrens von Gerth et al. [1] habe ich mit Hilfe von Isabelle/HOL [9] realisiert. Dabei ist Isabelle/HOL in erster Linie ein interaktiver Theorembeweiser. Die Abkürzung HOL steht für Higher Order Logic [10] und bezeichnet die Logik höherer Stufe. Die Bezeichnung Isabelle/HOL soll dabei andeuten, dass Isabelle im Kontext der Logik höherer Stufe verwendet wird. Tatsächlich ist Isabelle modular aufgebaut und erlaubt es, verschiedene Logiken zu verwenden. HOL eignet sich für die Zwecke der Implementierung eines Verfahrens nicht nur deswegen, weil ein Codegenerator zur Verfügung steht, der bestimmte HOL-Konstrukte in eine funktionale Sprache abbilden kann, sondern auch, weil viele Datenstrukturen, wie Listen, Zahlen etc., in HOL bereits verfügbar sind. Die Spezifikationssprache von Isabelle/HOL ist sehr ausdrucksstark und ist angelehnt an die Logik, wie man sie aus der Mathematik her kennt. HOL Konstrukte wie `datatype`-, `record`- oder `function`-Definitionen dagegen kennt man so von funktionalen Programmiersprachen.

4.1 Typen

Ähnlich wie viele funktionale Programmiersprachen basiert Isabelle/HOL auf dem getypten λ-Kalkül [15]. Deswegen hat in Isabelle/HOL jeder Ausdruck, der im eigentlichen Sinne einem λ-Term[1] entspricht, einen Typ. Neben den Basistypen wie `bool` (Wahrheitswerte) und `nat` (natürliche Zahlen) existiert in Isa-

[1] Funktionen können als λ-Terme aufgefasst werden. Die Inkrementierungsfunktion lässt sich etwa durch $\mathsf{inc} := \lambda n.n + 1$ definieren.

belle/HOL der Funktionstyp (gekennzeichnet durch ⇒), der totale[2] Funktionen repräsentiert. Ebenfalls unterstützt werden polymorphe Typen, die es ermöglichen mit einer Typvariable (als `'a` gekennzeichnet) über Typen zu abstrahieren.

In Isabelle/HOL existieren verschiedene Konstrukte, um Typen zu definieren. Mit dem Schlüsselwort `types` lassen sich bereits bekannte Typen durch einen neuen Namen abkürzen. Wenn man etwa eindeutige Bezeichner durch natürliche Zahlen repräsentieren möchte, so bietet es sich, der besseren Lesbarkeit wegen, einen neuen Namen für den Typ dieser Bezeichner einzuführen.

```
types
  id = nat
```

Zwar wird der Typ `id` Isabelle-intern weiterhin als `nat` behandelt, für den Leser erschließt sich jedoch auf diese Weise der Verwendungszweck von `id` eher.

4.1.1 Induktive Datentypen

Mit dem Schlüsselbegriff `datatype` ist man in der Lage, einen induktiven Datentypen zu definieren. Dabei ist die Definition eines induktiven Datentyps analog zu dem Konzept, das auch in der funktionalen Programmierung zum Einsatz kommt. Ein beliebtes Beispiel sind Listen[3].

```
datatype
  'a list = Nil                    ("[]")
          | Cons 'a "'a list"      (infixr "#" 65)
```

Der Ausdruck `Nil` stellt die leere Liste dar. Eine Liste, die das Element `0` enthält, erhält man durch `Cons 0 Nil`. Zusätzlich zur eigentlichen Datentypdefinition lässt sich ad hoc eine alternative Syntax definieren. Statt der Konstruktorschreibweise für die Liste mit dem Element `0` etwa ist man so in der Lage eine wohl bekannte Syntax zu verwenden, die dann diesen Ausdruck durch `0#[]`[4] abkürzen lässt. Gleichzeitig legt die Annotation **infixr** fest, dass sich der Listenkonstruktor `#` rechtsassoziativ (mit einer Priorität von `65`) zu verhalten hat. Im Kontext der

[2] Eine Funktion (im Sinne der Informatik) heißt total, falls sie jedem Element aus dem Definitionsbereich genau ein Bildelement zuordnet. Tatsächlich entspricht die Definition einer totalen Funktion der Definition einer Funktion im mathematischen Sinne.

[3] Listen sind bereits in Isabelle/HOL vordefiniert.

[4] In Isabelle/HOL ist es außerdem möglich, den Ausdruck `0#[]` zu `[0]` zu reduzieren.

Codegenerierung ist schließlich erwähnenswert, dass Datentypdefinitionen beim Abbilden in eine funktionale Sprache nahezu unverändert übertragen werden, da das `datatype`-Konzept von Isabelle/HOL dem der funktionalen Sprachen ähnelt.

4.1.2 Records

Das `records`-Konzept [11, Kap. 8.3] in Isabelle/HOL verallgemeinert das Prinzip von Tupeln[5] und ermöglicht die einzelnen Felder mit Namen anstatt mit der Position zu adressieren. Bei größeren Strukturen mit mehr als zwei Feldern profitiert man primär von der besseren Lesbarkeit des `records`-Konzepts. Durch Vergabe aussagekräftiger Feldnamen erhöht man zudem das intuitive Verständnis der Definition. Ein einfaches Beispiel soll dies verdeutlichen.

```
record book =
  title :: string
  author :: string
  pages :: nat
```

Diese offensichtliche Repräsentation eines Buches bedarf nahezu keiner Erklärung, da die Feldnamen den eigentlichen Einsatzzweck dieser Definition bereits verdeutlichen. Ein Ausdruck vom Typ `book` lässt sich mit der in Isabelle/HOL eingebauten Syntax wie folgt definieren:

```
abbreviation my_book :: book
where
  "my_book ≡ (| title = ''Harry Potter'',
                author = ''Joanne K. Rowling',
                pages =  334 |)"
```

Gleichzeitig kann man erkennen, wie in Isabelle/HOL durch **abbreviation** Abkürzungen definiert werden können. Wird die Typdekoration, hier gegeben durch `:: book`, weggelassen, versucht Isabelle den Typen durch Typinferenz automatisch zu ermitteln. Der Record selbst ist durch den Ausdruck in Klammern (| |) gegeben. Um etwa auf das `title`-Feld von `my_book` zugreifen zu können, schreibt man einfach `title my_book`. Eine Aktualisierung des Feldes `pages` dagegen erhält man durch `my_book(| pages := 335 |)`.

[5] Tupeln sind ebenfalls in Isabelle/HOL vordefiniert.

4.1.3 Sonstige Typen

In Isabelle/HOL existieren viele bereits vordefinierte Typen. Im Kontext dieser Arbeit sind vor allem Listen, wie sie im Abschnitt 4.1.1 vorgestellt wurden, von Interesse. Listen werden im Kontext der Codegenerierung verwendet, um endliche Mengen zu modellieren. Nichtsdestotrotz kann es aus einer theoretischen Betrachtung heraus einfacher sein, eine Aussage über Mengen anstatt über Listen zu formulieren. Mengen werden in Isabelle/HOL als Funktionen vom Typ `'a ⇒ bool` realisiert. Unabhängig davon ist Isabelle/HOL um eine entsprechend intuitive Syntax und viele nützliche Definitionen für Mengen angereichert. Das Bild einer Menge `A` unter einer Funktion `f` etwa hat man durch:

```
f ‘ A = {y. ∃x∈A. y = f(x)}
```

Ebenfalls vordefiniert sind übliche Operationen auf Mengen wie Vereinigung (∪), Schnitt (∩) und Differenz (-). Der Typ `'a set` repräsentiert eine Menge von Elementen vom Typ `'a`. Nicht zu verwechseln ist der Typkonstruktor `set` mit der Funktion `set`, die eine Liste in eine Menge überführt, die exakt die Listenelemente enthält.

Für Listen existieren eine Reihe nützlicher Funktionen, die bereits vordefiniert sind. Die Infixfunktion `@`, auch bekannt unter dem Namen `concat`, verbindet zwei Listen miteinander. Die Funktion `filter` liefert für die Parameter `P` und `xs` eine Liste zurück, die aus Elementen von `xs` besteht, für die das Prädikat `P` erfüllt ist. Für `filter P xs` bietet Isabelle/HOL ebenfalls eine alternative Syntax: `[x←xs. P x]`. Mit der Funktion `length` lässt sich die Länge einer Liste bestimmen. Ist eine natürliche Zahl `i` kleiner als die Länge einer Liste `xs`, so erhält man durch `xs!i` das `i`-te Element dieser Liste. Schließlich entfernt die Funktion `remdups` aus einer Liste Elemente, die mehrfach vorkommen.

4.2 Funktionen

In Isabelle/HOL beginnen Funktionsdefinitionen [12], wie man sie von funktionalen Sprachen her kennt, mit dem Schlüsselwort **fun** bzw. **function**. Das folgende Beispiel definiert die Ackermannfunktion:

```
fun ack :: "(nat × nat) ⇒ nat"
where
```

```
  "ack(0, m) = Suc m"
| "ack(Suc n, 0) = ack(n, 1)"
| "ack(Suc n, Suc m) = ack(n, ack(Suc n, m))"
```

Der Bezeichner `ack` stellt dabei den Namen der Funktion dar. Hinter den zwei Doppelpunkten `::` folgt der Typ der Funktion. Demnach handelt es sich bei `ack` um eine Funktion mit einem Paar natürlicher Zahlen als Argument und einer natürlichen Zahl als Rückgabewert. Die Definition[6] der Funktion selbst ist gegeben durch eine Reihe von Gleichungen, die mit dem Zeichen `|` getrennt werden. Wie man dem Beispiel entnehmen kann, wird das Pattern Matching auf Termen eines Datentyps unterstützt. Oberflächlich gesehen unterscheidet sich das `function`-Konzept von dem der funktionalen Programmiersprachen kaum. Tatsächlich versucht Isabelle/HOL eine Reihe von Eigenschaften automatisch zu überprüfen wie etwa, dass die definierte Funktion stets terminiert oder dass das Pattern Matching vollständig ist. Da in Isabelle/HOL alle Funktionen total sein müssen, um die Konsistenz des Systems zu erhalten, ist die Überprüfung der Terminierungseigenschaft zwingend notwendig. Eine Funktion, die auf bestimmten Werten nicht terminiert, kann aber nicht total sein, da sie diesen Werten keine Elemente aus dem Bildbereich zuordnet. Schlägt die automatische Überprüfung fehl, so wird die Definition abgewiesen. Das kann zwei Ursachen haben: Entweder die Definition erfüllt tatsächlich nicht die Forderungen, die an eine Funktion gestellt werden, oder die automatische Methode war nicht in der Lage diese nachzuweisen. Trifft Letzteres zu, so bleibt nur noch die Möglichkeit diese Eigenschaften von Hand zu verifizieren. In diesem Fall verwendet man statt **fun** das Schlüsselwort **function**, um eine Funktionsdefinition einzuleiten. Tatsächlich handelt es sich bei einer **fun**-Definition um ein Konstrukt, das auf **function** basiert.

Nicht-rekursive Funktionen können auch durch **definition** eingeführt werden, wie das nachfolgende Beispiel zeigt.

```
definition list_inc :: "[nat list, nat] ⇒ nat list"
where
  "list_inc ns n ≡ map (op + n) ns"
```

Die so definierte Funktion `list_inc` besitzt zwei Argumente, in der Reihenfolge wie in den Klammern `[ ]` angegeben. Das Ergebnis der Funktion wird durch das Anwenden der Funktion `op + n` (im Wesentlich identisch zu `f(m) = n + m`) auf jedes einzelne Element der Liste `ns` ermittelt.

[6] Die Funktion `Suc` berechnet den Nachfolger einer natürlichen Zahl. Dabei ist diese Funktion so definiert, das sie auch als Datentypkonstruktor betrachtet werden kann.

4.3 Aussagen und Beweise

Die Essenz von Isabelle/HOL besteht darin, Aussagen formalisieren und beweisen zu können. Mit den Schlüsselwörter **lemma** und **theorem** kann man eine zu zeigende Aussage notieren. Der Beweis dieser Aussage erfolgt dann durch Anwenden bereits bekannter Aussagen mit Hilfe spezieller Beweismethoden, die Isabelle/HOL zur Verfügung stellt. Während die größtenteils sehr umfangreichen Isabelle-Beweise in dieser schriftlichen Ausarbeitung nicht wiedergegeben werden[7], wird die Isabelle-Notation für Aussagen immer wieder verwendet werden. Aus diesem Grund soll diese mit dem folgenden Beispiel verdeutlicht werden.

```
lemma subset_elem:
  assumes "x ∈ A"
      and "A ⊆ B"
    shows "x ∈ B"
```

Wie bereits erwähnt, beginnt die Einleitung einer Aussage mit dem Schlüsselwort **lemma**. Der Bezeichner `subset_elem` referenziert dieses Lemma, um etwa die Aussage im Beweis einer anderen Aussage verwenden zu können. Mit **assumes** wird die Prämisse dieses Lemmas eingeleitet und mit **and** mit weiteren Annahmen verknüpft. Die aus dieser Voraussetzung zu folgernde Aussage wird schließlich mittels **shows** notiert.

Beispiel Folgendes Beispiel liefert einen Eindruck, wie eine Aussage in Isabelle formalisiert und bewiesen werden kann. Bei der Aussage handelt es sich um die Expansionsstrategie für eine Until-Formel, die in Abschnitt 3.1.2 vorgestellt wurde.

```
lemma ltl_expand_Until:
  shows "ξ ⊨ φ U ψ ⟷ ξ ⊨ ψ or (φ and X (φ U ψ))"
     (is "?lhs = ?rhs")
proof
  assume ?lhs
  then obtain i
       where psi_is: "suffix i ξ ⊨ ψ"
        and phi_is: "∀j<i. suffix j ξ ⊨ φ" by auto
```

[7] Rechtfertigungen einer Aussage werden stets durch eine kleine Beweisskizze gegeben sein.

```
  show ?rhs
  proof(cases i)
    assume "i = 0"
    thus ?rhs using psi_is by auto
  next
    fix k
    assume i_eq: "i = Suc k"
    with phi_is have "ξ ⊨ φ" by auto
    moreover
    have "ξ ⊨ X (φ U ψ)" using psi_is phi_is i_eq by auto
    ultimately show ?rhs by auto
  qed
next
  assume rhs: ?rhs
  show ?lhs
  proof(cases "ξ ⊨ ψ")
    assume "ξ ⊨ ψ"
    then have "suffix 0 ξ ⊨ ψ" by auto
    moreover
    have "∀j<0. suffix j ξ ⊨ φ" by auto
    ultimately
    have "∃i. suffix i ξ ⊨ ψ
              ∧ (∀j<i. suffix j ξ ⊨ φ)" by blast
    thus ?lhs by auto
  next
    assume "¬ ξ ⊨ ψ"
    then have phi_is: "ξ ⊨ φ"
          and "ξ ⊨ X (φ U ψ)" using rhs by auto
    then obtain i
        where psi_suc_is: "suffix (Suc i) ξ ⊨ ψ"
          and phi_suc_is: "∀j<i. suffix (Suc j) ξ ⊨ φ" by auto
    have "∀j<(Suc i). suffix j ξ ⊨ φ"
    proof(clarify)
      fix j
      assume j_less: "j<Suc i"
      show "suffix j ξ ⊨ φ"
      proof (cases j)
        assume "j = 0"
        thus ?thesis using phi_is by auto
```

```
      next
        fix k
        assume "j = Suc k"
        thus ?thesis using j_less phi_suc_is by auto
      qed
    qed
    thus ?lhs using psi_suc_is phi_is by auto
  qed
qed
```

Im Wesentlichen handelt es sich bei diesem Beweisskript um eine Dekomposition der ursprünglichen Aussage in mehrere kleinere Teilaussagen, die mit Hilfe spezieller Methoden, wie etwa der Fallunterscheidung (`cases`), bewiesen werden. Viele Beweisschritte lassen sich auch automatisch herleiten, sodass diese dann nicht explizit ausgeschrieben werden müssen. Um die Lesbarkeit solcher Beweise zu erhöhen, empfiehlt es sich jedoch, für das Verständnis entscheidende Zwischenschritte explizit auszuformulieren.

Das hier vorgestellt Beweisskript soll nur einen Eindruck vermitteln, wie Beweise mit Isabelle/HOL angefertigt werden können. Für ein tiefer gehendes Verständnis solcher Beweisskripte ist das Referenzhandbuch von Isabelle [11] hilfreich.

Kapitel 5

Isabelle-Implementierung des Verfahrens

In diesem Kapitel stelle ich meine Implementierung des Verfahrens von Gerth et al. [1] zur Konstruktion eines Büchi-Automaten aus einer LTL-Formel, wie es in Kapitel 3 kurz skizziert wurde, vor. Im Zuge dessen gehe ich detailliert auf das Verfahren ein und erläutere dessen Funktionsweise genau. Zunächst stelle ich die Datenstrukturen vor, die für die Konstruktion des Graphen (vgl. Abschnitt 3.1) notwendig sind, um damit anschließend die Funktion, die den Graphen konstruiert, zu definieren. Daraufhin zeige ich, dass diese Funktion stets terminiert. Im Anschluss gehe ich auf die Transformation des Graphen in einen LGBA ein und zeige, dass dieser LGBA die entsprechenden Korrektheitseigenschaften erfüllt.

Die Implementierung des Verfahrens ist stets unter dem Aspekt der Codegenerierung zu betrachten, denn um Code aus einer Isabelle-Definition generieren zu können, müssen bestimmte Voraussetzungen (vgl. Kapitel 6) erfüllt werden.

5.1 Modellierung von LTL-Formeln

5.1.1 Syntax für LTL-Formeln

Um LTL-Formeln gemäß Abschnitt 2.2 definieren zu können, benötigt man zunächst eine Repräsentation für Propositionen.

```
types
  "prop" = string
```

Dazu führe ich zunächst eine neue Typabkürzung `prop`[1] ein. Eine Proposition wird folglich durch einen String repräsentiert. Diese Wahl eignet sich gut für die Zwecke der Codegenerierung. Das generierte Programm soll am Ende ausgeführt werden können. Dazu muss die Eingabeformel des Benutzers, die als String vorliegt, in einer geeigneten Art und Weise in die Isabelle-Repräsentation transformiert werden (vgl. Abschnitt 6.4). Mit der Repräsentation der Propositionen als String können diese direkt abgebildet werden.

Die LTL-Formeln selbst werden mit dem `datatype`-Package (vgl. Abschnitt 4.1.1) implementiert.

```
datatype
  frml = LTLTrue             ("true")
       | LTLFalse            ("false")
       | LTLProp "prop"      ("prop'(_')")
       | LTLNeg frml         ("not _" [84] 84)
       | LTLAnd frml frml    (infixr "and" 83)
       | LTLOr frml frml     (infixr "or" 82)
       | LTLNext frml        ("X _" [84] 84)
       | LTLUntil frml frml  (infixr "U" 81)
       | LTLUDual frml frml  (infixr "V" 80)
```

Somit erhält man einen neuen Typen `frml`, der gerade LTL-Formeln repräsentiert. Die Notation in Klammern bietet eine Möglichkeit, für die Termkonstruktoren eine alternative Schreibweise[2] festzulegen. Dabei ist das Zeichen _ ein Platzhalter für die Argumente des jeweiligen Termkonstruktors. Statt etwa den Ausdruck „`LTLOr LTLTrue LTLFalse`" hin zuschreiben, kann man diesen auch in der schöneren Syntax als „`true or false`" notieren. Die Verwendung dieser schöneren Schreibweise beschränkt sich allerdings auf die Theoriedateien[3] von Isabelle. Für die Codegenerierung hat sie keine Bedeutung (vgl. Kapitel 6).

Im Gegensatz zu Abschnitt 2.2 werden in der Definition der LTL-Formeln die Formeln `true`, `false`, „φ `and` ψ" und „φ `V` ψ nicht als Abkürzungen, sondern direkt als LTL-Formeln eingeführt. Dadurch hat man den Vorteil, dass Funktionen auf LTL-Formeln für alle relevanten Fälle mittels Pattern Matching definiert werden können (vgl. Abschnitt 5.1.2).

[1] Die Anführungszeichen in der Definition müssen gesetzt werden, da das Wort `prop` bereits an einer anderen Stelle verwendet wird.

[2] Das Symbol ' schützt Zeichen in einer Syntaxdefinition, die standardmäßig eine andere Funktion haben.

[3] Definitionen und Beweise werden in Isabelle in Theoriedateien organisiert. Das Konzept ähnelt dem modularen Aufbau von funktionalen Programmen.

5.1.2 Semantik für LTL-Formeln

Um die Semantik für LTL-Formeln definieren zu können, müssen ω-Wörter ebenfalls in Isabelle/HOL implementiert[4] werden. Ein ω-Wort über dem Alphabet A kann als eine Funktion von den natürlichen Zahlen in die Menge A betrachtet werden. Die Isabelle/HOL-Implementierung für ω-Wörter sieht demnach wie folgt aus:

```
types
  'a word = nat ⇒ 'a
```

Das zugrunde liegende Alphabet A wird durch die Menge der Elemente des Typs `'a` repräsentiert. Sei ein ω-Wort `w` vom Typ „`'a word`" gegeben. Da es sich bei `w` um eine Funktion auf den natürlichen Zahlen handelt, liefert `w(i)`[5] gerade das `i`-te Element von `w`. Das `k`-Suffix von `w` erhält man durch die Funktion „λ`n. w(k+n)`", abgekürzt durch „`suffix k w`".

Mit der folgenden Typdeklaration wird eine Interpretation gemäß Abschnitt 2.2 modelliert.

```
types
  interprt = "(prop set) word"
```

Eine Interpretation ist folglich ein ω-Wort über dem Alphabet, das aus Mengen von Propositionen besteht.

Um die Semantik zu definieren, verwende ich das `function`-Package (vgl. Abschnitt 4.2). Dabei wird die Eingabeformel rekursiv in die logische Sprache von HOL übersetzt.

```
fun semantics :: "[interprt, frml] ⇒ bool" ("_ ⊨ _" [80,80] 80)
where
  "ξ ⊨ true = True"
| "ξ ⊨ false = False"
| "ξ ⊨ prop(q) = (q∈ξ(0))"
| "ξ ⊨ not φ = (¬ ξ ⊨ φ)"
| "ξ ⊨ φ and ψ = (ξ ⊨ φ ∧ ξ ⊨ ψ)"
```

[4] Die Isabelle-Theorien über ω-Wörter wurden zur Verfügung gestellt von Stephan Merz.
[5] Der Ausdruck "`w(i)` wird intern mit "`w i` identifiziert.

```
| "ξ ⊨ φ or ψ = (ξ ⊨ φ ∨ ξ ⊨ ψ)"
| "ξ ⊨ X φ = (suffix 1 ξ ⊨ φ)"
| "ξ ⊨ φ U ψ = (∃i. suffix i ξ ⊨ ψ ∧ (∀j<i. suffix j ξ ⊨ φ))"
| "ξ ⊨ φ V ψ = (∀i. suffix i ξ ⊨ ψ ∨ (∃j<i. suffix j ξ ⊨ φ))"
```

Die Funktion `semantics` ordnet einer Interpretation und einer LTL-Formel einen Wahrheitswert zu. Dabei werden die linken Seiten obiger Gleichungen durch die rechten Seiten definiert. Für die erste Gleichung gilt etwa, dass der Ausdruck „$\xi \models$ `true`" dem Wahrheitswert `True` zugeordnet wird. Dabei ist dieser Ausdruck nur eine andere Schreibweise für "`semantics` ξ `true`".

Die Semantik selbst ist exakt so definiert wie in Abschnitt 2.2. Die alternative Syntax ermöglicht es dabei, sich auch optisch der Definition in Abschnitt 2.2 anzunähern. Für die neu hinzugekommenen Fälle `true`, `false`, „φ `and` ψ" und „φ `V` ψ", die in Abschnitt 2.2 als Abkürzungen eingeführt wurden, muss die Semantik explizit definiert werden. Um die Tatsache zu untermauern, dass die hier definierte Semantik tatsächlich der in Abschnitt 2.2 entspricht, bietet es sich an, diese Behauptung für die Abkürzungen formal nachzuweisen. Relativ einfach einzusehen ist die semantische Identität für `true` und `false`.

```
lemma ltl_true_or_con:
  shows "ξ ⊨ true ⟷ ξ ⊨ prop(p) or (not prop(p))"

lemma ltl_false_true_con:
  shows "ξ ⊨ false ⟷ ξ ⊨ not true"
```

Die Gültigkeit der beiden Gleichungen folgt direkt aus der Definition der Semantik.

Der duale Zusammenhang zwischen „φ `or` ψ" und „φ `and` ψ" sowie zwischen „φ `U` ψ" und „φ `V` ψ" lässt sich ebenfalls relativ leicht verifizieren. Das erledigt für die letztere Aussage das folgende Lemma:

```
lemma ltl_UDual_Until_con:
  shows "ξ ⊨ φ V ψ ⟷ ¬ ξ ⊨ (not φ) U (not ψ)"
```

Um diese einfache Aussage zu zeigen, reicht es, sich die Definition der Semantik anzuschauen. Durch einfache Umformungen auf der logischen Ebene von HOL erhält man dann die Aussage direkt.

Die Konstruktion des Graphen aus Abschnitt 3.1 basiert auf der Beobachtung aus Abschnitt 3.1.2, jede LTL-Formel semantisch äquivalent so umformulieren zu können, dass ein Teil dieser Formel unmittelbar und ein anderer Teil im nächsten Zustand erfüllt ist. Diese Expansionsstrategie ist für Boolesche Operatoren und den Next-Operator leicht nachvollziehbar. Für die Operatoren `U` und `V` lassen sich folgende Identitäten nachweisen:

lemma `ltl_expand_Until:`
  shows "$\xi \models \varphi$ `U` $\psi \longleftrightarrow \xi \models \psi$ `or (`φ `and X (`φ `U` ψ`))`"

lemma `ltl_expand_UDual:`
  shows "$\xi \models \varphi$ `V` $\psi \longleftrightarrow \xi \models \psi$ `and (`φ `or X (`φ `V` ψ`))`"

Der Beweis des Lemmas `ltl_expand_UDual` gestaltet sich einfach, wenn man bereits `ltl_expand_Until` gezeigt hat. Schließlich kann man den dualen Bezug von `U` und `V` verwenden, der in `ltl_UDual_Until_con` hergeleitet wurde, um die Aussage zu verifizieren. Um jedoch `ltl_expand_Until` zu beweisen[6], betrachtet man die Definition der Semantik von „φ `U` ψ" und macht eine Fallunterscheidung darüber, ob ψ unmittelbar erfüllt ist oder nicht.

5.1.3 Übergang zur Negationsnormalform

Die Konstruktion des Graphen basiert auf der Annahme, dass die Eingabeformel in Negationsnormalform vorliegt. Um eine LTL-Formel in Negationsnormalform zu bringen, definiere ich folgende Funktion:

```
fun pushneg :: "frml ⇒ frml"
where
  "pushneg true = true"
| "pushneg false = false"
| "pushneg prop(q) = prop(q)"
| "pushneg (not true) = false"
| "pushneg (not false) = true"
| "pushneg (not prop(q)) = not prop(q)"
| "pushneg (not (not ψ)) = pushneg ψ"
| "pushneg (not (ν and μ)) = pushneg (not ν) or pushneg (not μ)"
| "pushneg (not (ν or μ)) = pushneg (not ν) and pushneg (not μ)"
| "pushneg (not (X ψ)) = X pushneg (not ψ)"
```

[6] In Abschnitt 4.3 wird diese Aussage im Stile von Isabelle/HOL verifiziert.

```
| "pushneg (not (ν U μ)) = pushneg (not ν) V pushneg (not μ)"
| "pushneg (not (ν V μ)) = pushneg (not ν) U pushneg (not μ)"
| "pushneg (φ and ψ) = (pushneg φ) and (pushneg ψ)"
| "pushneg (φ or ψ) = (pushneg φ) or (pushneg ψ)"
| "pushneg (X φ) = X (pushneg φ)"
| "pushneg (φ U ψ) = (pushneg φ) U (pushneg ψ)"
| "pushneg (φ V ψ) = (pushneg φ) V (pushneg ψ)"
```

Die Funktion `pushneg` liefert Negationsnormalform: Um zu begründen, dass die durch die Funktion `pushneg` resultierende LTL-Formel tatsächlich in Negationsnormalform vorliegt, ist der Begriff der Teilformeln erforderlich, denn der Definition 3.1.1 der Negationsnormalform nach muss man dazu sinngemäß jede negierte Teilformel der ursprünglichen LTL-Formel betrachten. Dazu definiere ich zunächst induktiv die direkte Teilformel-Beziehung.

```
inductive is_subfrml_step :: "[frml, frml] ⇒ bool"
where
  "is_subfrml_step φ (not φ)"
| "is_subfrml_step φ (φ and ψ)"
| "is_subfrml_step ψ (φ and ψ)"
| "is_subfrml_step φ (φ or ψ)"
| "is_subfrml_step ψ (φ or ψ)"
| "is_subfrml_step φ (X φ)"
| "is_subfrml_step φ (φ U ψ)"
| "is_subfrml_step ψ (φ U ψ)"
| "is_subfrml_step φ (φ V ψ)"
| "is_subfrml_step ψ (φ V ψ)"
```

Die Formel φ steht zu ψ in der direkten Teilformel-Beziehung, falls die Aussage „`is_subfrml_step` φ ψ" erfüllt ist. Der reflexiv-transitive Abschluss dieser Beziehung liefert schließlich die allgemeine Teilformel-Beziehung.

```
abbreviation
  is_subfrml :: "[frml, frml] ⇒ bool" (infixr "is'_subfrml" 80)
where
  "ψ is_subfrml(φ) ≡ is_subfrml_step** ψ φ"
```

Für ein zweistelliges Prädikat „`p :: ['a, 'a] => bool`" erhält man durch `p`** gerade den reflexiv-transitiven Abschluss von `p`, d. h. es gilt:

- `p`** `x x` für alle `x` des Typs `'a`[7];
- wenn `p`** `x y` und `p y z` erfüllt ist, dann auch `p`** `x z`.

Mit „ψ `is_subfrml(`φ`)`" wird die Aussage, dass ψ eine Teilformel von φ ist, formuliert. Das nachfolgende Theorem zeigt schließlich, dass die Funktion `pushneg` tatsächlich eine LTL-Formel in Negationsnormalform liefert.

```
theorem pushneg_struct:
  assumes "(not ψ) is_subfrml(pushneg φ)"
    shows "∃q. ψ = prop(q)"
```

Um dieses Theorem zu beweisen, kann man wie folgt vorgehen. Zunächst zeigt man per Induktion über die Struktur von φ das folgende Lemma:

```
lemma pushneg_neg_struct:
  assumes "pushneg φ = not ψ"
    shows "∃q. ψ = prop(q)"
```

Dieses besagt, dass, wenn eine durch `pushneg` transformierte Formel mit einem Negationszeichen beginnt, diese Formel dann eine negierte Proposition sein muss. Andererseits zeigt man, dass eine Teilformel einer durch `pushneg` transformierten Formel wiederum eine solche Form besitzen muss.

```
lemma subformula_on_pushneg:
  assumes "ψ is_subfrml(pushneg φ)"
    shows "∃μ. ψ = pushneg μ"
```

Dieses Lemma zeigt man ebenfalls induktiv über φ. Mit diesen zwei Aussagen folgert man dann das Theorem direkt.

Die Funktion `pushneg` erhält die Semantik: Wie das folgende Theorem zeigt, bleibt unter der `pushneg`-Transformation die semantische Bedeutung der Eingabeformel erhalten.

[7] Hier ist `'a` eine Typvariable, an deren Stelle ein beliebiger Typ stehen kann (vgl. Abschnitt 4.1).

```
theorem pushneg_equiv:
  shows "ξ ⊨ φ ⟷ ξ ⊨ pushneg φ"
```

Diese wichtige Aussage lässt sich relativ einfach, per Induktion über φ nachweisen.

Mit `pushneg` existiert also eine Funktion, die eine beliebige LTL-Formel in eine LTL-Formel in Negationsnormalform überführt, die dabei die gleiche Semantik besitzt wie die ursprüngliche Formel.

Unter dem Aspekt der Transformation stellt sich die Frage, wie groß die generierte Ausgabeformel gegenüber der Eingabe ist. Wenn die Ausgabeformel gegenüber der Eingabe etwa exponentiell anwachsen würde, so könnte man den Sinn und Zweck dieser Transformation in Frage stellen. Glücklicherweise ist das nicht der Fall, da sich eine obere Schranke für die Größe der Formel wie folgt angeben lässt:

```
theorem pushneg_size_lin:
  shows "size (pushneg φ) ≤ 2 * size φ"
```

Somit ist die Ausgabe von `pushneg` höchstens zwei mal so groß[8] wie die Eingabeformel. Um diese Aussage zu verifizieren, kann man die Eigenschaft verwenden, dass `pushneg` die Anzahl der logischen Operatoren, das Negationszeichen ausgenommen, nicht verändert. Wenn nun diese gerade konstant bleibt und das Hineinziehen des Negationszeichens bewirkt, dass es schließlich vor jeder Proposition der ursprünglichen Formel auftaucht, so kann man folgern, dass sich die transformierte Formel höchstens um die Anzahl der Propositionen vergrößert hat. Damit folgt die eigentliche Behauptung direkt.

5.2 Graphenkonstruktion

Das Verfahren aus Kapitel 3, das es nun zu implementieren gilt, erfordert, dass die Eingabeformel in Negationsnormalform vorliegt. In 5.1.3 habe ich gezeigt, dass jede beliebige LTL-Formel mit Hilfe von `pushneg` in eine LTL-Formel in Negationsnormalform überführt werden kann. Unter diesem Aspekt liegt es nahe, die Implementierung so festzulegen, dass die Eingabeformel vor der Verarbeitung durch das Verfahren von `pushneg` transformiert wird. Das Verfahren selbst wird über der Struktur der LTL-Formeln definiert. Da nun sich die Struktur der

[8] Die Funktion `size` zählt alle logischen Symbole, die in einer LTL-Formel enthalten sind.

LTL-Formeln, so wie in Abschnitt 3.1.1 beschrieben, beim Übergang in die Negationsnormalform ändert, erscheint es unpraktisch, die `datatype`-Definition aus Abschnitt 5.1 zu verwenden. Angenommen man möchte mit dieser `datatype`-Definition eine Funktion über der Struktur von LTL-Formeln in Negationsnormalform definieren, etwa eine Funktion, die die Anzahl der Literale[9] zählt. So könnte die Funktionsdefinition aussehen:

```
fun leafcnt :: "frml ⇒ nat"
where
  "leafcnt true = 1"
| "leafcnt false = 1"
| "leafcnt prop(q) = 1"
| "leafcnt (not φ) = leafcnt φ"
| "leafcnt (φ and ψ) = (leafcnt φ) + (leafcnt ψ)"
| "leafcnt (φ or ψ) = (leafcnt φ) + (leafcnt ψ)"
| "leafcnt (X φ) = leafcnt φ"
| "leafcnt (φ U ψ) = (leafcnt φ) + (leafcnt ψ)"
| "leafcnt (φ V ψ) = (leafcnt φ) + (leafcnt ψ)"
```

Die Definition an sich ist unproblematisch. Nur, wenn man davon ausgehen könnte, dass der Funktion `leafcnt` als Eingabe nur LTL-Formeln in Negationsnormalform übergeben werden, würde sich der allgemeine Negationsfall „`not` φ" auf den speziellen Negationsfall „`not prop(q)`" reduzieren lassen. In obiger Funktionsdefinition ist das jedoch nicht möglich, da es keinerlei Information darüber gibt, dass die Eingabeformel in Negationsnormalform vorliegt.

Um dieses Problem zu umgehen, habe ich mich entschieden, für LTL-Formeln in Negationsnormalform einen eigenen Datentyp[10] zu definieren.

5.2.1 LTL-Formeln in Negationsnormalform

Die Datentypdefinition für LTL-Formeln in Negationsnormalform sieht wie folgt aus:

```
datatype
  frml = LTLTrue_n                ("true_n")
```

[9] Ein Literal ist eine Proposition, die LTL-Formel `true` oder die LTL-Formel `false`.

[10] Damit sind die Betrachtungen aus Abschnitt 5.1 zunächst irrelevant, werden aber in Kapitel 6 wieder aufgegriffen.

```
| LTLFalse_n                ("false_n")
| LTLProp_n "prop"          ("prop_n'(_')")
| LTLNProp_n "prop"         ("nprop_n'(_')")
| LTLAnd_n frml frml        (infixr "and_n" 83)
| LTLOr_n frml frml         (infixr "or_n" 82)
| LTLNext_n frml            ("X_n _" [84] 84)
| LTLUntil_n frml frml      (infixr "U_n" 81)
| LTLUntilDual_n frml frml (infixr "V_n" 80)
```

Das Subskript $_n$ in der Syntax und das `_n` an den Konstruktoren sollen die Negationsnormalform andeuten. Wie bereits erwähnt, unterscheidet sich diese `datatype`-Definition von der in Abschnitt 5.1 dadurch, dass es nur negierte Propositionen statt negierten LTL-Formeln gibt.

5.2.2 Zustandsrepräsentation

Für den Graphen, der gemäß Abschnitt 3.1 konstruiert wird, ist eine entsprechende Repräsentation erforderlich. Hier wird der Graph als eine Liste von Zuständen modelliert, dessen Transitionen in den Zuständen gespeichert sind. Jeder Zustand q hat dazu einen eindeutigen Namen sowie ein Feld, in dem alle Namen der Zustände enthalten sind, von denen eine Kante zu q führt. Zustandsnamen werden durch natürliche Zahlen modelliert.

```
types
  node_name = nat
```

Diese Repräsentation bietet sich deswegen an, weil man damit relativ einfach neue Zustandsnamen generieren kann und weil man nie in die Verlegenheit kommen kann, nicht mehr genügend freie Zustandsnamen zur Verfügung zu haben. Ein Zustand hat folgende Darstellung:

```
record node =
  name :: node_name
  incoming :: "node_name list"
  old :: "frml list"
  "next" :: "frml list"
```

Neben den zwei bereits angedeuteten Feldern `name` und `incoming` hat ein Zustand zwei weitere Felder `old` und `next`. Wie in Abschnitt 3.1 beschrieben, wird die Eingabeformel rekursiv in den Teil, der unmittelbar gilt, und in den Teil, der im nächsten Zustand erfüllt sein muss, zerlegt. Im Feld `next` werden gerade die Formeln abgelegt, die im nächsten Zustand erfüllt sein müssen, während in `old` alle Formeln abgelegt werden, die im Zuge des Zerlegungsprozesses in Betracht gezogen wurden. Demzufolge werden in `old` u. a. die (negierten) Propositionen abgelegt, die unmittelbar erfüllt sein müssen bzw. nicht erfüllt sein dürfen. Tatsächlich ist es so, dass die Expansionsstrategie, naiv angewendet, ggf. unendlich oft ausgeführt werden muss. Wenn man etwa die Identität $\xi \vDash \phi \mathsf{U} \psi \equiv \xi \vDash \psi \vee (\phi \wedge \mathsf{X}(\phi \mathsf{U} \psi))$ gemäß Lemma `ltl_expand_Until` genauer untersucht, so muss die Formel $\phi \mathsf{U} \psi$ unter bestimmten Voraussetzungen sowohl für den aktuellen als auch für den nächsten Zustand erfüllt sein. Dadurch könnte der Fall eintreten, dass man stets den gleichen Zerlegungsprozess durchführt und somit unendlich viele gleiche Zustände generiert. Um dies zu verhindern, wird ein Protokoll des Zerlegungsprozesses geführt, in dem die zerlegten Formeln entsprechend in die Felder `old` und `next` abgelegt werden. Folglich hat jeder fertiggestellter Zustand, eine Historie in sich gespeichert, wie er aus der Eingabeformel hervorging. Trifft es nun zu, dass man während der Verarbeitung den exakt gleichen Zerlegungsprozess wie bei einem bereits fertiggestellten Zustand durchgeht, so identifiziert man den aktuellen Zustand mit dem bereits konstruierten, anstatt ihn neu zu erzeugen.

5.2.3 Verfahrensdefinition

Die Definition der Funktion, die den Graphen konstruiert, ist unterteilt in die Funktion `expand`, die die Expansionsstrategie implementiert, und `create_graph`, die `expand` mit speziellen Startwerten, insbesondere mit der Eingabeformel, aufruft.

5.2.3.1 Repräsentation der Argumente

Die Funktion `expand` besitzt als Argument ein Paar bestehend aus einer Formelliste `fs` und einem Zustand `n`. Die Idee ist, in `fs` die Formeln zu speichern, die noch verarbeitet werden müssen, während `n` den aktuellen Automatenzustand repräsentiert, der während der Verarbeitung sukzessive konstruiert wird. Zu Beginn besteht `fs` nur aus der Eingabeformel. Hat die Eingabeformel etwa die Gestalt $\phi \wedge \psi$, dann könnte sowohl ϕ als auch ψ eine Aussage beinhalten, die den aktuellen Zustand betrifft. Aus diesem Grund müssen beide Formeln bzgl. ihrer

Struktur analysiert, in die Formelliste eingefügt und von `expand` rekursiv weiterverarbeitet werden. Gleichzeitig wird die Eingabeformel als verarbeitet betrachtet. Angenommen die Eingabeformel hat die Gestalt $\mathsf{X}\phi$, so wird die Formel ϕ für den nächsten Zustand vorgemerkt. Auf den aktuellen Zustand hat die Formel $\mathsf{X}\phi$ keinen Einfluss, muss demnach auch nicht in die Formelliste des Arguments eingefügt werden.

Die Repräsentation für das Paar-Argument von `expand` ist wie folgt definiert:

```
types
  cnode = "frml list * node"
```

Die erste Komponente vom Typ „`frml list`" wird als `new`-Komponente (vgl. Abschnitt 3.1.2) bezeichnet, um anzudeuten, dass es sich dabei um Formeln handelt, die noch zu verarbeiten sind. Die zweite Komponente heißt `node`-Komponente und modelliert den Zustand, der während der rekursiven Verarbeitung der Eingabeformel sukzessive konstruiert wird. Dieser Zustand gilt dann als fertiggestellt, wenn die Formelliste leer ist, also keine zu analysierende Formeln mehr enthält. Folgende zwei Abkürzungen stellen sicher, dass die beiden Bezeichner `new` und `node` entsprechend definiert sind.

```
abbreviation new :: "cnode ⇒ frml list"
where
  "new ≡ fst"

abbreviation node :: "cnode ⇒ node"
where
  "node ≡ snd"
```

Diese zwei Abkürzungen verwenden die bereits vordefinierten Funktion `fst` und `snd`, die jeweils die erste bzw. die zweite Komponente eines Paares liefern. Diese Vorgehensweise ähnelt dem `records`-Konzept aus Abschnitt 4.1.2. Folglich ließe sich `cnode` auch als Record darstellen. Der Vorteil der Paar-Repräsentation von `cnode` (vgl. Abschnitt 5.2.4) besteht jedoch darin, dass man bei Funktionsdefinitionen auf Argumenten des Typs `cnode` Pattern Matching verwenden kann.

5.2.3.2 Hilfsfunktionen

Die Expansionsstrategie für Formeln der Form $\mu\mathsf{U}\psi$, $\mu\mathsf{V}\psi$ und $\mu\vee\psi$ erzeugt unter Umständen eine Verzweigung. Ausgehend von $\mu \vee \psi$ ist es klar, dass es so sein

muss, da der von `expand` konstruierte Graph gerade die Pfade modellieren soll, auf denen die Eingabeformel erfüllt ist. Hat die Eingabeformel nun die Gestalt $\mu \vee \psi$, so müssen sowohl Pfade für μ als auch für ψ analysiert werden. Wenn nicht gerade μ und ψ identisch sind oder mindestens eine der beiden Formeln unerfüllbar ist, liegt es nahe, zwei verschiedene Zustände erzeugen zu müssen. Die Expansionsstrategie für die anderen Fälle liefert die Identitäten $\xi \vDash \mu \mathsf{U} \psi \equiv \xi \vDash \psi \vee (\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi))$ bzw. $\xi \vDash \mu \mathsf{V} \psi \equiv \xi \vDash \psi \wedge (\mu \vee \mathsf{X}(\mu \mathsf{V} \psi))$, die ebenfalls eine Oder-Aussage beinhaltet. Letztere entspricht mittels logischer Umformung folgender Äquivalenz:

$$\xi \vDash \mu \mathsf{V} \psi \equiv \xi \vDash (\psi \wedge \mu) \vee (\psi \wedge \mathsf{X}(\mu \mathsf{V} \psi))$$

Analog zum Fall der Disjunktion müssen für den V-Fall ggf. auch zwei Zustände generiert werden, nur diesmal für die Formeln $\psi \wedge \mu$ und $\psi \wedge \mathsf{X}(\mu \mathsf{V} \psi)$. Auf diese Weise extrahieren die folgenden Hilfsfunktionen (vgl. Abschnitt 3.1.2) für die Fälle $\mu \mathsf{U} \psi$, $\mu \mathsf{V} \psi$ und $\mu \vee \psi$ die notwendigen Information, um eine Verzweigung des Graphen einzuleiten.

```
fun new1 :: "frml ⇒ frml list"
where
  "new1 (μ Un ψ)  = [μ]"
| "new1 (μ Vn ψ)  = [ψ]"
| "new1 (μ orn ψ) = [μ]"

fun new2 :: "frml ⇒ frml list"
where
  "new2 (μ Un ψ)  = [ψ]"
| "new2 (μ Vn ψ)  = [μ, ψ]"
| "new2 (μ orn ψ) = [ψ]"

fun next1 :: "frml ⇒ frml list"
where
  "next1 (μ Un ψ)  = [μ Un ψ]"
| "next1 (μ Vn ψ)  = [μ Vn ψ]"
| "next1 (μ orn ψ) = []"

fun neg :: "frml ⇒ frml"
where
  "neg propn(q)  = npropn(q)"
```

```
| "neg nprop_n(q) = prop_n(q)"
```

Die Funktionen `new1` und `next1` berechnen gemäß der Expansionsstrategie für den ersten Teil der Disjunktion die Formeln, die noch im aktuellen Zustand zu analysieren sind, und die Formeln, die im nächsten Zustand analysiert werden. Im Falle von $\mu \mathsf{V} \psi$ erhält man gerade aus dem rechten Disjunktionsglied von $(\psi \wedge \mu) \vee (\psi \wedge \mathsf{X}(\mu \mathsf{V} \psi))$ die Forderung, ψ im aktuellen und $\mu \mathsf{V} \psi$ im nächsten Zustand analysieren zu müssen. Die Funktion `new2` liefert analog die Formeln für den zweiten Teil der Disjunktion. Für $\mu \mathsf{V} \psi$ hat man also die Forderung, dass sowohl ψ als auch μ im aktuellen Zustand noch zu analysieren sind. Eine Funktion `next2` ist dabei nicht erforderlich, da der zweite Teil der Disjunktion keine explizite Aussage über den nächsten Zustand beinhaltet.

Es fällt auf, dass die Hilfsfunktionen zwar als Argument eine beliebige LTL-Formel in Negationsnormalform akzeptieren, jedoch nur auf einigen wenigen Fällen davon definiert sind. Isabelle handhabt das so, dass, wenn etwa `new1` auf $\texttt{true}_n$ angewendet wird, die durch „`new1 true`$_n$" resultierende LTL-Formel einfach nicht bekannt ist. Folglich kann man für „`new1 true`$_n$" keinerlei Eigenschaften nachweisen, die für LTL-Formeln in der Regel gelten. Überträgt man diese Hilfsfunktionen mittels Codegenerierung in eine funktionale Programmiersprache wie OCaml [14] und kompiliert anschließend den resultierenden Quellcode, so erhält man Hinweise[11] darauf, dass die Hilfsfunktionen auf bestimmten Werten nicht definiert sind. Doch gerade diese Hinweise können ignoriert werden, wenn innerhalb von Isabelle nachgewiesen werden kann, dass diese Funktionen nie auf Formeln angewandt werden, auf denen sie nicht definiert sind. Dies geschieht hier dadurch, indem die Korrektheit der Konstruktion in Abschnitt 5.3.3 nachgewiesen wird.

Die Funktion `expand` verwendet die Hilfsfunktion `upd_nds`. Diese ist wie folgt definiert:

```
definition upd_nds
where
  "upd_nds P ns n
   = map (λx. if P n x then
              x(| incoming := remdups((incoming n)@(incoming x)) |)
              else x) ns"
```

[11] Solche Hinweise (this pattern-matching is not exhaustive) beklagen sinngemäß die fehlende Vollständigkeit des Pattern Matchings.

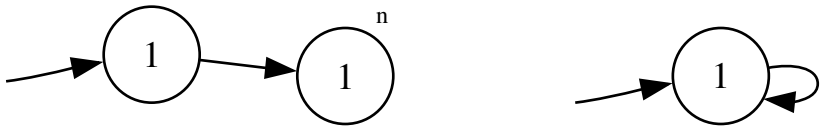

Abbildung 5.1: Identifikation von Zuständen

Die Funktion `upd_nds` aktualisiert das `incoming`-Feld bereits konstruierter Knoten, die eine bestimme Bedingung `P` erfüllen. Dabei ist diese Bedingung in der Definition von `expand` gerade so gewählt, dass `upd_nds` genau die Zustände aus `ns` aktualisiert, die die gleiche Historie wie `n` besitzen, d. h. deren `old`- und `next`-Felder mit denen von `n` übereinstimmen. Trifft diese Bedingung auf einen Zustand aus `ns` zu, so ist dieser in gewisser Hinsicht gleich zu `n`, da dieser aus einer LTL-Formel hervorgegangen ist, die zu der LTL-Formel, aus der `n` hervorging, semantisch gleich sein muss. Nichtsdestotrotz können sich beide Zustände im `incoming`-Feld unterscheiden. Um die Transitionen, die zu `n` führen, nicht zu verlieren, ist die Zusammenführung der `incoming`-Felder von `n` und des bereits existierenden Zustandes notwendig. Wie im Beispiel aus Abbildung 5.1 angedeutet, wird ein fertig konstruierter Zustand, der wie oben beschrieben zu einem anderen bereits konstruierten Zustand gleich ist, nicht neu erzeugt, sondern mit diesem identifiziert. Die eingehende Kante wird dabei entsprechend umgebogen.

5.2.3.3 Eindeutigkeit von Namen

Die Transitionen des konstruierten Graphen werden dadurch beschrieben, dass ein Zustandsname des Zustands, von dem die Transition ausgeht, im `incoming`-Feld des eingehenden Zustands abgelegt wird. Dafür ist es erforderlich, jeden Zustand anhand seines Namens eindeutig identifizieren zu können. Ein Zustand gilt außerdem als Startzustand, falls der ausgezeichnete Name init, der mit keinem Zustandsnamen identisch sein sollte, im `incoming`-Feld des Zustandes enthalten ist. Zustandsnamen werden hier durch natürliche Zahlen repräsentiert. Deshalb bietet es sich an, init wie folgt zu definieren:

```
constdefs
  init :: node_name "init ≡ 0"
```

Demzufolge wird die Konstante `init` vom Typ `node_name` mit 0 identifiziert. Die Funktion, die den Graphen konstruiert, muss dabei sicherstellen, dass alle vergebenen Zustandsnamen ungleich 0 sind.

Die Konstruktion des Graphen erfolgt schrittweise. Folglich lässt sich die Zuteilung von Zustandsnamen ebenfalls schrittweise realisieren. Jedes mal wenn ein Zustand fertiggestellt wird und die Verarbeitung eines neuen Zustands beginnt, reicht es aus, den Zustandsnamen des fertiggestellten Zustands um 1 zu erhöhen, um so einen neuen Namen für den neuen Zustand zu generieren.

```
abbreviation new_name :: "node ⇒ node_name"
where
  "new_name n ≡ Suc (name n)"
```

Die Funktion `new_name` erzeugt gerade ausgehend von einem Zustand einen neuen Namen. Da die Zuteilung von Namen sukzessive erfolgt, ist die Eindeutigkeit dieser Zustandsnamen gewährleistet.

Schließlich wird es erforderlich sein, ausgehend von einem rekursiven Aufruf den nächsten freien Zustandsnamen zu ermitteln. Das geschieht hier dadurch, indem der Rückgabewert von `expand` um so einen Namen erweitert wird.

```
record expand_rslt =
  nextname :: node_name
  nodes  :: "node list"
```

Neben der eigentlichen Ausgabe, der Liste von Zuständen, liefert `expand` nun zusätzlich den nächsten freien Zustandsnamen zurück.

5.2.3.4 Implementierung

Im Folgenden stelle ich die Implementierung von `expand` vor. Um auf jeden einzelnen Fall separat eingehen zu können, werde ich die Fälle, die sich aus dem Pattern Matching auf dem Paar-Argument (vgl. Abschnitt 5.2.3.1) ergeben, getrennt von einander betrachten.

```
function (sequential) expand :: "[cnode, node list] ⇒ expand_rslt"
where
```

Die Funktionsdefinition wird mit dem Schlüsselwort `function` gefolgt von einer Reihe in Klammern festgelegter Optionen eingeleitet. Hier verwende ich die Option `sequential`, die es ermöglicht, Pattern Matching im Sinne funktionaler Programmiersprachen zu benutzen.[12] Der festgelegte Typ von `expand` besagt, dass `expand` eine Funktion mit zwei Argumenten ist, die einen Wert vom Typ `expand_rslt` zurück gibt. Beim ersten Argument handelt es sich um ein Paar bestehend aus einer `new-` und einer `node`-Komponente, so wie in Abschnitt 5.2.3 beschrieben. Das zweite Argument ist die Ausgabeliste, die von Aufruf zu Aufruf durchgegeben und ggf. erweitert wird, um sie schließlich, sobald die Berechnung stoppt, auszugeben.

Der erste Fall der Funktionsdefinition von `expand` ist durch die Bedingung gegeben, dass die `new`-Komponente des ersten Arguments leer ist.

```
"expand ([], n) ns
 = (if (∃nd∈set ns. set (old nd) = set (old n) ∧
                    set (next nd) = set (next n))
    then (| nextname = name n,
            nodes = upd_nds (λn nd. set (old nd) = set (old n) ∧
                                    set (next nd) = set (next n)) ns n |)
    else expand (next n,
                 (|name = new_name n,
                  incoming = [name n],
                  old = [],
                  next = [] |)) (n#ns))"
```

Da das zweite Argument `ns` die bereits fertiggestellten Zustände enthält, prüft die `if`-Bedingung, ob dort ein Zustand mit der gleichen Historie wie die des gerade fertig konstruierten Zustandes `n` existiert. Die Historie eines Zustandes wird in dessen `old-` und `next`-Feldern kodiert. Ist die `if`-Bedingung erfüllt, so wird der Zustand `n` mit dem übereinstimmenden Zustand mittels `upd_nds` (vgl. Abschnitt 5.2.3.2) identifiziert, andernfalls wird die Analyse weitergeführt. Zwar ist die `new`-Komponente leer, d. h. der Analysevorgang für den aktuellen Zustand ist abgeschlossen, das `next`-Feld des Zustandes `n` enthält jedoch ggf. zu analysierende Formeln. Folglich muss der Analysevorgang für den nächsten Zustand initiiert werden. Dazu wird `expand` rekursiv aufgerufen. Die `new`-Komponente des ersten Arguments enthält gerade die Formeln, die im nächsten Zustand zu analysieren sind, also die Formeln aus `next n`. Der Zustand, mit dem die Berechnung beginnt, besitzt einen noch unbenutzten Namen „`new_name n`" (vgl. Abschnitt 5.2.3.3) sowie eine von `n` eingehende Kante. Diese Kante wird dadurch repräsentiert, dass

[12] Darauf werde ich im letzten Fall der `expand`-Definition näher eingehen.

der Name von `n` im `incoming`-Feld des neuen Zustandes abgelegt wird. Die Felder `old` und `next` sind folgerichtig leer, da dort erst Formeln abgelegt werden, wenn sie der `new`-Komponente entnommen und entsprechend verarbeitet wurden. Das zweite Argument vergrößert sich um den Zustand `n`, da dieser Zustand nun als fertiggestellt gilt.

In den nächsten beiden Fällen besteht die Situation, in der eine (negierte) Proposition analysiert werden muss.

```
| "expand (prop_n(q)#fs, n) ns
   = (if neg prop_n(q) ∈ set (old n) then (| nextname = name n, nodes = ns |)
      else expand (fs,
                   n(| old := [φ←[prop_n(q)]. φ ∉ set(old n)] @ old n |)) ns)"
| "expand (nprop_n(q)#fs, n) ns
   = (if neg nprop_n(q) ∈ set (old n) then (| nextname = name n, nodes = ns |)
      else expand (fs,
                   n(| old := [φ←[nprop_n(q)]. φ ∉ set(old n)] @ old n |)) ns)"
```

In beiden Fällen wird eine Überprüfung, ob die Negation der (negierten) Proposition in dem `old`-Feld von `n` enthalten ist oder nicht, durchgeführt. Da im `old`-Feld von `n` gerade alle bisher verarbeiteten Formeln abgelegt werden, also u. a. die Propositionen, die im aktuellen Zustand erfüllt sein müssen, lässt sich so ein einfacher Konsistenztest realisieren. Wenn sowohl die (negierte) Proposition als auch die Negation davon in einem Zustand gleichzeitig erfüllt sein müssen, führt dies dazu, dass dieser Zustand widersprüchlich ist. Dieser inkonsistente Zustand sowie dessen Folgezustände müssen nicht betrachtet werden, da sonst der so konstruierte Graph Pfade besitzen würde, für die kein Modell einer erfüllbaren Formel existieren könnte. Trifft die `if`-Bedingung nicht zu, so wird der Analysevorgang rekursiv für die Restliste `fs` fortgeführt. Zusätzlich wird die (negierte) Proposition in das `old`-Feld des aktuellen Zustands aufgenommen, falls diese nicht schon darin enthalten ist.

Der nächste Fall betrifft die LTL-Formel $\top$.

```
| "expand (true_n#fs, n) ns
   = expand (fs, n(| old := [φ←[true_n]. φ ∉ set(old n)] @ old n |)) ns"
```

Die LTL-Formel $\top$ stellt keinerlei Forderungen an den aktuellen Zustand, da sie bekanntlich stets erfüllt ist. Folglich kann sie aus der `new`-Komponente entfernt werden, um rekursiv den Analysevorgang fort zu führen.

Der nächste Fall betrifft die LTL-Formel $\bot$.

```
| "expand (false_n#fs, n) ns
   = (| nextname = name n, nodes = ns |)"
```

In diesem Fall hat man erneut eine Forderung für den aktuellen Zustand, die nie erfüllt werden kann. Folglich müssen keinerlei Anstrengungen unternommen werden, um diesen Zustand weiter zu verarbeiten.

Im nächsten Fall wird eine Konjunktion betrachtet.

```
| "expand (μ and_n ψ#fs, n) ns
   = expand ([φ←[μ,ψ]. φ ∉ set (fs@old n)] @ fs,
             n(| old := [φ←[μ and_n ψ]. φ ∉ set(old n)] @ old n |)) ns"
```

Sowohl das linke als auch das rechte Konjunktionsglied könnten Aussagen über den aktuellen und über den nächsten Zustand beinhalten. Folglich müssen diese rekursiv weiterverarbeitet werden, falls dies nicht bereits vorher geschehen ist. Dazu werden beide LTL-Formeln in die `new`-Komponente des rekursiven Arguments abgelegt.

Der nächste Fall betrifft die Next-Formel.

```
| "expand (X_n μ#fs, n) ns
   = expand (fs,
             n(| old := [φ←[X_n μ]. φ ∉ set(old n)] @ old n,
                next := [φ←[μ]. φ ∉ set(next n)] @ next n|)) ns"
```

Dieser Fall trägt nichts zum aktuellen Zustand bei. Folglich kann die Next-Formel aus der `new`-Komponente komplett entfernt werden. Dagegen muss die Formel μ im nächsten Zustand erfüllt sein und wird folgerichtig in das `next`-Feld eingefügt.

Die restlichen drei Fälle Disjunktion, Until und Release werden durch einen gemeinsamen letzten Fall verarbeitet. Hierfür sind die drei bereits definierten Hilfsfunktionen `new1`, `new2` und `next1` erforderlich. Ebenfalls zum Einsatz kommt verschachtelte Rekursion.

```
| "expand (f#fs, n) ns
   = (let rslt
          = expand ([φ←new1 f. φ ∉ set (fs@old n)] @ fs,
                    n(| old := [φ←[f]. φ ∉ set(old n)] @ old n,
                       next := [φ←next1 f. φ ∉ set (next n)] @ next n |)) ns
      in expand ([φ←new2 f. φ ∉ set (fs@old n)] @ fs,
                 n(| name := nextname rslt,
                    old := [φ←[f]. φ ∉ set(old n)] @ old n |)) (nodes rslt))"
```

Wie bereits erwähnt, muss in diesem Fall ggf. eine Verzweigung konstruiert werden. Die Idee ist nahe verwandt mit dem „Divide & Conquer"-Prinzip, welches ein Problem in mehrere kleinere Probleme aufteilt, die kleineren Probleme löst und die berechneten Lösungen auf eine gewissen Art und Weise zu einer Gesamtlösung zusammenfügt. Im Falle der Disjunktion wählt das Verfahren das linke Glied der Disjunktion und berechnet die Lösung für diese Teilformel. Von dieser Teillösung aus, die durch den inneren Aufruf berechnet wird, geht das Verfahren sinngemäß wieder an die ursprüngliche Stelle des Graphen zurück und vervollständigt den Graphen bzgl. des rechten Glieds der Disjunktion durch den äußeren Aufruf von `expand`. Ausgehend von der bereits vorgestellten Expansionsstrategie verwendet das Verfahren die Eigenschaft, dass sich die Formeln $\mu \mathsf{U} \psi$ bzw. $\mu \mathsf{V} \psi$ zu den Disjunktionen $\psi \vee (\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi))$ bzw. $(\psi \wedge \mu) \vee (\psi \wedge \mathsf{X}(\mu \mathsf{V} \psi))$ umformen lassen. Die Konstruktion des Graphen für die Formeln $\mu \mathsf{U} \psi$ bzw. $\mu \mathsf{V} \psi$ kann so auf die Konstruktion bzgl. einer Disjunktion zurückgeführt werden. Da die Struktur dieser Umformungen im Vorfeld bekannt ist, wird die Aufteilung in `new` und `next` an dieser Stelle explizit vorgenommen. Genau diese Aufteilung wird mit den Hilfsfunktionen `new1`, `new2` und `next1` realisiert.

Ausgehend von der rekursiven Funktion `expand` folgt nun die Definition der Funktion `create_graph`.

```
abbreviation start_node
where
  "start_node φ ≡ ([φ], ⦇ name = 1,
                          incoming = [init],
                          old = [],
                          next = [] ⦈)"

definition create_graph :: "frml ⇒ node list"
where
  "create_graph φ ≡ nodes (expand (start_node φ) [])"
```

Die Funktion `create_graph` ruft die Funktion `expand` mit Startwerten auf. Die `new`-Komponente des ersten Arguments enthält nur die Eingabeformel φ. Die Felder `old` und `next` der `node`-Komponente enthalten keine Formeln, da zum Zeitpunkt des Aufrufs die Eingabeformel noch nicht verarbeitet wurde. Das `incoming`-Feld enthält den Namen `init`, um festzulegen, dass die Konstruktion des Graphen mit einem Startzustand beginnt. Das zweite Argument, das die Liste der fertiggestellten Zustände repräsentiert, ist zu Beginn leer. Der Zustandsname des Startzustands ist gerade so gewählt, dass er mit `init` nicht kollidiert.

Da alle nachfolgend zugeteilten Zustandsnamen sukzessive ermittelt werden, sind auch diese zu `init` verschieden.

5.2.3.5 Induzierter Graph

Sei `ns` das Ergebnis von `create_graph` φ. Die Funktion `create_graph` definiert einen Graphen `(V,`Δ`)` im Sinne von Kapitel 3 wie folgt:

- `V = set ns`;
- Δ`(n) = set [n'`$\leftarrow$`ns. name n'` $\in$ `set (incoming n)]`.

5.2.4 Konstruktiver Vergleich

Dieser Abschnitt beschäftigt sich mit dem Vergleich der Graphenkonstruktion aus Abschnitt 3.1.2 gemäß Gerth et al. [1] und meiner Implementierung dieses Verfahrens gemäß Abschnitt 5.2.3.

Der Algorithmus zur Konstruktion des Graphen aus einer LTL-Formel nach Gerth et al. [1] aus Abbildung 3.3 liegt in Pseudo-Code vor. Folglich kann ein konstruktiver Vergleich gegenüber einer Implementierung des Verfahrens nur bedingt vorgenommen werden, da Pseudo-Code-Konstrukte in der Regel (absichtlich) allgemein gehalten werden. Deswegen sind die Unterschiede gegenüber meiner Implementierung in den meisten Fällen technischer Natur.

Bereits die Zustandsrepräsentation (Abbildung 3.3, Zeile 1-2) unterscheidet sich technisch bedingt von der Implementierung. Um Pattern Matching auf dem Eingabeargument von `expand` zu ermöglichen, habe ich das `new`-Feld von der Zustandsrepräsentation abgekoppelt (vgl. Abschnitt 5.2.3.1). Die einzige nichttechnische Änderung in diesem Zusammenhang ist der Wegfall des *Father*-Feldes. Ursprünglich verwendet die Beweisidee von Gerth et al. [1] dieses Feld, um bestimmte Korrektheitseigenschaften (im Wesentlichen Lemma 4.5 aus Abschnitt 5.3.3.4 und Lemma 4.7 aus Abschnitt 5.3.3.4) herzuleiten. Da diese Eigenschaften jedoch auch ohne die Argumentation mit Hilfe des *Father*-Feldes bewiesen werden können, ist die Notwendigkeit dieses zu modellieren, nicht mehr vorhanden. Ebenfalls unterscheidet sich die Repräsentation der Bezeichner. Während der Pseudo-Code diese als Strings modelliert, verwende ich natürliche Zahlen als Namensrepräsentation.

Ebenfalls technisch bedingt werden Mengen durch Listen repräsentiert. Aus der Isabelle-Spezifikation des Verfahrens gemäß Abschnitt 5.2.3 soll Code generiert

(vgl. Kapitel 6) werden. Da der Codegenerator von Isabelle Mengen nicht direkt in eine funktionale Sprache abbilden kann, ist es erforderlich, eine Repräsentation zu wählen, bei der dies möglich ist.

Die Implementierung des Verfahrens selbst unterscheidet sich prinzipiell kaum von der ursprünglichen Vorlage. Die Rolle des obersten if-then-else-Konstrukts (Abbildung 3.3, Zeile Zeile 4, Zeile 11) sowie die des case-Konstrukts (Abbildung 3.3, Zeile 14) übernimmt das Pattern Matching. Dabei werden die Fälle, die sich auf Literale beziehen, explizit dargestellt. Die if-Abfrage und der dazugehörige then-Teil im Fall des leeren *New*-Feldes (Abbildung 3.3, Zeile 5-10) sind in Pseudo-Code jedoch derartig allgemein gehalten, dass sich die Implementierung zwingenderweise davon unterscheiden muss. Um diesen Teil des Verfahrens effizient zu gestalten, wäre es sogar denkbar, das gesamten if-then-else-Konstrukt umzuschreiben. Die Erweiterung des Rückgabewertes von `expand` gegenüber der Pseudo-Code-Darstellung ist eine Folge der Namenszuteilung, die Zustandsnamen ohne Seiteneffekte vergibt. Beim Erzeugen einer Verzweigung (Abbildung 3.3, Zeile 20-27) ist es außerdem nicht erforderlich beiden Zuständen (*Node1*, *Node2*) einen neuen Namen zu vergeben, da der erste Folgezustand denselben Namen wie der Ausgangszustand erhalten kann.

Die Funktionsweise des Verfahrens ist folglich nahezu gleich in die Implementierung eingebettet. Änderungen, die sich gegenüber der Pseudo-Code-Darstellung ergeben haben, sind überwiegend technischer Natur.

5.2.5 Beweis der Terminierung

In diesem Abschnitt geht es darum, zu zeigen, dass das in Abschnitt 5.2.3 beschriebene Verfahren terminiert. Den Kern des Verfahrens stellt die rekursive Funktion `expand` dar, die von `create_graph` mit speziellen Werten aufgerufen wird. Um die Terminierung von `create_graph` zu zeigen, reicht es aus, die Terminierung der rekursiven Funktion `expand` nachzuweisen.

Die rekursive Funktion `expand` ist gegeben durch eine Reihe von Gleichungen. Dabei wird die linke Seite dieser Gleichungen durch die rechte Seite definiert. Auf der rechten Seite kann ein Aufruf von `expand` vorkommen. Ausgehend vom Startaufruf von `expand` in der Definition von `create_graph` muss die linke Seite genau[13] einer dieser Gleichungen dazu passen. Beim Berechnen dieses Aufrufs wird dieser durch die rechte Seite der passenden Gleichung ersetzt. Kommt in

[13] In Isabelle müssen die Pattern einer `function`-Definition paarweise verschieden und vollständig sein. In der Definition von `expand` sorgt dafür die Option `sequential`.

dieser rechten Seite wieder ein Aufruf von `expand` vor, so wiederholt sich diese Prozedur. Lässt sich nun nachweisen, dass sich diese Expansion der Definition von `expand` nur endlich oft wiederholt, so kann man daraus folgern, dass die Berechnung in diesem Fall terminiert. Um zu zeigen, dass diese Folge von rekursiven Aufrufen endlich ist, betrachtet man die Argumente der Funktion. Weist man nach, dass die Argumente einer Folge von Aufrufen in jedem Schritt bzgl. einer wohlfundierten Ordnung fallen, so folgt daraus die Endlichkeit der Folge von Aufrufen. Die Lösung des eigentlichen Problems besteht also darin, eine passende wohlfundierte Ordnung zu finden, die die soeben beschriebenen Eigenschaften erfüllt.

Betrachtet man nun die linken Seiten der Gleichungen, die die Funktion `expand` definieren, so stellt man fest, dass diese Gleichungen bzgl. der `new`-Komponente des ersten Arguments in zwei Kategorien zerfallen. Man hat den Fall, in dem die `new`-Komponente leer ist, sowie die Fälle, in denen die `new`-Komponenten nicht leer sind. Letztere beschreiben die Konstruktion des aktuellen Zustands, während der Fall der leeren `new`-Komponente gerade einen Zustandsübergang konstruiert bzw. die Konstruktion des aktuellen Zustands abschließt.

5.2.5.1 Konstruktion des aktuellen Zustands

Ausgehend von der Definition von `expand` stellt man nun für die Fälle, in denen die `new`-Komponente des ersten Arguments nicht leer ist, fest, dass die `new`-Komponenten des Arguments der rekursiven Aufrufe von `expand` in einer gewissen Art und Weise kleiner werden und zwar wie folgt:

`new`-Komponente (links)	`new`-Komponente (rechts)
$\texttt{prop}_n\texttt{(q)\#fs}$	`fs`
$\texttt{nprop}_n\texttt{(q)\#fs}$	`fs`
$\texttt{true}_n\texttt{\#fs}$	`fs`
$\texttt{X}_n\ \mu\texttt{\#fs}$	`fs`

Für diese vier Fälle dekrementiert die `new`-Komponente ausgehend von den Argumenten auf der linken Seite gegenüber den Argumenten der rechten Seite einer Gleichung bzgl. der Listenlänge. Durch das Herausnehmen des obersten Elements der Eingabeliste wird die resultierende Liste bzgl. der Anzahl ihrer Elemente echt kleiner.

Für den $\texttt{false}_n\texttt{\#fs}$-Fall gilt, dass kein rekursiver Aufruf von `expand` in der rechten Seite der entsprechenden Gleichung vorkommt. Folglich kann dieser Fall aus

dieser Betrachtung herausgenommen werden.

Im Falle der Konjunktion ergibt sich folgender Bezug der `new`-Komponenten:

`new`-Komponente (links)	`new`-Komponente (rechts)
μ `and`$_n$ ψ`#fs`	`[`φ`←[`μ`,`ψ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`

Hier kann man nicht mit der Anzahl der Elemente argumentieren, denn die Anzahl der Listenelemente in der `new`-Komponente (rechts) könnte größer sein, als die der `new`-Komponente (links). Ähnliches gilt für die restlichen Fälle.

`new`-Komponente (links)	`new`-Komponente (rechts)
μ `or`$_n$ ψ`#fs`	`[`φ`←[`μ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`
μ `or`$_n$ ψ`#fs`	`[`φ`←[`ψ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`
μ `U`$_n$ ψ`#fs`	`[`φ`←[`μ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`
μ `U`$_n$ ψ`#fs`	`[`φ`←[`ψ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`
μ `V`$_n$ ψ`#fs`	`[`φ`←[`ψ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`
μ `V`$_n$ ψ`#fs`	`[`φ`←[`μ`,`ψ`]. `$\varphi \notin$ `set (fs@old n)] @ fs`

Pro Fall sind hier jeweils zwei `new`-Komponenten auf der rechten Seite der entsprechenden Gleichung zu berücksichtigen, da die rechte Seite dieser Gleichung zwei Aufrufe von `expand` enthält. Wie bereits festgestellt, kann die Anzahl der Listenelemente nicht zur Definition einer fallenden Ordnung herangezogen werden. Betrachtet man jedoch die strukturelle Größe[14] einer Formel, so stellt man fest, dass diese für das oberste Element der `new`-Komponente stets abnimmt. Entweder die Formel verschwindet komplett oder sie wird bzgl. ihrer Struktur zerlegt, d. h. man erhält nur Teilformeln der ursprünglichen Formel. Folgende Funktion berechnet die strukturelle Größe einer LTL-Formel.

```
fun size_frml :: "frml ⇒ nat"
where
  "size_frml (φ and_n ψ) = size_frml φ + size_frml ψ + 1"
| "size_frml (X_n φ) = size_frml φ + 1"
| "size_frml (φ U_n ψ) = size_frml φ + size_frml ψ + 1"
| "size_frml (φ V_n ψ) = size_frml φ + size_frml ψ + 1"
| "size_frml (φ or_n ψ) = size_frml φ + size_frml ψ + 1"
```

[14] Eine LTL-Formel besteht aus Konstruktoren wie `LTLTrue`, `LTLFalse` usw. Die strukturelle Größe einer LTL-Formel ist folglich durch die Anzahl der darin enthaltenen Konstruktoren gegeben.

```
| "size_frml _ = 1"
```

Diese Funktion gewichtet jeden Operator und jedes Literal mit dem Wert 1 und summiert dann diese Werte aufeinander. Entfernt man nun etwa aus einer LTL-Formel der Form „μ `and`$_n$ ψ" den Operator `and`$_n$, so ist die Summe der Größen der Teilformeln μ und ψ echt kleiner als die Größe der ursprünglichen LTL-Formel. Die nachfolgende Funktion ist die offensichtliche Verallgemeinerung der strukturellen Größe von LTL-Formeln auf Listen.

```
abbreviation size_frml_list :: "frml list ⇒ nat"
where
  "size_frml_list fs ≡ listsum (map size_frml fs)"
```

Ausgehend von einer Liste von LTL-Formeln berechnet diese Funktion zunächst die strukturelle Größe für jede einzelne LTL-Formel dieser Liste und summiert dann die Ergebnisse auf.

5.2.5.2 Konstruktion des Zustandsübergangs

Zwar hat man nun ausgehend von den bisherigen Betrachtungen eine Funktion, gemäß der die `new`-Komponenten der Argumente kleiner werden, jedoch gilt das nicht für jede Gleichung der Definition von `expand`. Im Fall, in dem die `new`-Komponente leer ist, d. h. in dem die Verarbeitung des aktuellen Zustands abgeschlossen ist, wird `expand` ggf. wieder rekursiv aufgerufen und zwar so, dass die `new`-Komponente des rekursiven Aufrufs ggf. größer als die des ursprünglichen Aufrufs werden kann. Zwar hat man mit der Funktion `size_frml_list` einen Teil der Lösung gefunden, das entscheidende Argument für die Terminierung von `expand` fehlt jedoch noch. Genau dieses Argument muss rechtfertigen können, dass das Verfahren nur endlich viele Zustände erzeugt. Das bedeutet im Sinne der Definition von `expand`, dass das zweite Argument durch eine obere Schranke begrenzt ist.

Beginnend mit dem Startaufruf von `expand`, der durch `create_graph` initiiert wird, konstruiert der Algorithmus jeden Zustand schrittweise. Dieser legt im Zuge der Berechnung Teilformeln der Eingabeformel in das `old`- und `next`-Feld des aktuellen Zustands ab. Ist die Konstruktion des Zustands, die, wie eben in Abschnitt 5.2.5.1 festgestellt, stets terminiert, abgeschlossen, d. h. befindet sich der Algorithmus in dem Fall, in dem die `new`-Komponente leer ist, so prüft er zunächst, ob bereits ein Zustand mit dem gleichen `old`- und `next`-Feld konstruiert

wurde. Ist das nicht der Fall, so fährt der Algorithmus mit der Konstruktion des nächsten Zustands fort. Wieder legt dieser im Zuge der Berechnung Teilformeln der Eingabeformel in das `old`- und `next`-Feld des Zustands ab und gelangt schließlich in die Situation, in der die `new`-Komponente erneut leer ist. Aufgrund der Tatsache, dass das `old`- und `next`-Feld eines Zustands stets Teilformeln der Eingabeformel beinhaltet, muss der Algorithmus spätestens dann abbrechen, wenn das zweite Argument, das die Liste der generierten Zustände enthält, gerade soviel Zustände umfasst, dass dadurch alle möglichen Kombination von verschiedenen `old`- und `next`-Feldern gegeben sind. Für eine LTL-Formel existieren nur endlich viele Teilformeln, sodass es auch nur endlich viele solcher Kombinationen geben kann.

Zunächst stellt sich die Frage, wie man eine obere Schranke für das zweite Argument von `expand` definieren könnte. Dazu erscheint es praktischer, statt den Zuständen selbst nur die relevanten Informationen, also deren `old`- und `next`-Felder, zu betrachten. Für diesen Zweck definiere ich den folgenden Begriff, der einen Zustand auf eine für die hier benötigten Zwecke geeignete Repräsentation abbildet.

```
abbreviation old_next_pair :: "node ⇒ (frml set × frml set)"
where
  "old_next_pair n ≡ (set (old n), set (next n))"
```

Ausgehend von einem Zustand erhält man durch `old_next_pair` ein Paar von Mengen, dessen Komponenten gerade die Formeln der `old`- und `next`-Felder des Zustands enthalten. Der Ausdruck „`old_next_pair ‘ set ns`" liefert für `ns`, das zweite Argument von `expand`, die Menge solcher Paare. Es ist insofern ausreichend, Mengen von Formeln zu betrachten, als die `if`-Bedingung im Fall, in dem die `new`-Komponente leer ist, sich ebenfalls auf die Gleichheit der `old`- und `next`-Felder als Mengen bezieht. Trifft diese `if`-Bedingung für die `node`-Komponente von `n` nicht zu, so vergrößert sich das zweite Argument des rekursiven Aufrufs von `expand`, gegeben durch `n#ns`, gemäß `old_next_pair`, wie das folgende Lemma zeigt.

```
lemma neg_if_cond_transf:
  assumes "∀nd∈set ns. set (old nd) = set (old n)
           ⟶ set (next nd) ≠ set (next n)"
    shows "old_next_pair n ∉ old_next_pair ‘ set ns"
```

Unter der Annahme der negierten `if`-Bedingung gilt gerade, dass kein Zustand in `ns` enthalten ist, der mit dem Zustand `n` in den `old`- und `next`-Feldern übereinstimmt. Folglich muss auch gelten, dass das dazugehörige Paar nicht in der Menge „`old_next_pair ‘ (set ns)`" enthalten ist. Aus dieser Folgerung ergibt sich nun die Eigenschaft, dass das zweite Argument im Falle der leeren `new`-Komponente echt anwächst, d. h. es gilt:

```
old_next_pair ‘ set ns ⊂ old_next_pair ‘ set (n#ns)
```

Für die anderen Fälle der Definition von `expand` gilt, dass entweder das zweite Argument konstant bleibt oder, wie im Falle der verschachtelten Rekursion, das zweite Argument potentiell größer sein könnte. In jedem Fall wird das zweite Argument nie kleiner.

Lässt sich nun für das zweite Argument eine obere Schranke `LIM` finden, so muss gemäß der Teilmengenbeziehung für den Fall der leeren `new`-Komponente folgende Aussage zutreffen.

```
LIM - old_next_pair ‘ set (n#ns) ⊂ LIM - old_next_pair ‘ set ns
```

Damit fällt das zweite rekursive Argument von `expand` bzgl. dieser Mengenbeziehung in diesem Fall, während in den anderen Fällen das rekursive Argument höchstens gleich bleibt.

Um die Schranke `LIM` definieren zu können, benötigt man alle möglichen Kombinationen von Paaren von `old`- und `next`-Mengen, die der Algorithmus ausgehend von beliebigen Anfangsargumenten im Stande ist, zu berechnen. Da dieser im Zuge der Berechnung Teilformeln der `new`-Komponente in die `old`- und `next`-Felder der `node`-Komponente ablegt, müssen diese Teilformeln in der Definition von `LIM` in irgendeiner Weise berücksichtigt werden. Dazu benötigt man zunächst den Begriff der Teilformeln für LTL-Formeln in Negationsnormalform. Im Gegensatz zum Begriff der Teilformeln für allgemeine LTL-Formeln aus Abschnitt 5.1.3, in dem ein Prädikat definiert wird, das entscheidet, ob eine LTL-Formel eine Teilformel einer anderen LTL-Formel ist, definiere ich diesen Begriff für LTL-Formeln in Negationsnormalform so, dass von einer LTL-Formel ϕ alle Teilformeln von ϕ berechnet werden.

```
fun subfrmls :: "frml ⇒ frml set"
where
  "subfrmls φ
```

```
= (case φ of
        μ and_n ψ ⇒ {φ} ∪ subfrmls μ ∪ subfrmls ψ
      | X_n μ ⇒ {φ} ∪ subfrmls μ
      | μ U_n ψ ⇒ {φ} ∪ subfrmls μ ∪ subfrmls ψ
      | μ V_n ψ ⇒ {φ} ∪ subfrmls μ ∪ subfrmls ψ
      | μ or_n ψ ⇒ {φ} ∪ subfrmls μ ∪ subfrmls ψ
      | _ ⇒ {φ}
    )"
```

Dabei besteht die Formelmenge „`subfrmls` φ" aus allen echten Teilformeln von φ sowie φ selbst. Basierend auf diesem Begriff definiere ich nun eine Menge, die alle im ersten Argument von `expand` vorkommende (Teil-)Formeln beinhaltet.

```
definition all_subfrmls :: "cnode ⇒ frml set"
where
  "all_subfrmls n
   ≡ ⋃subfrmls ‘ set (new n@old (node n)@next (node n))"
```

Für die Fälle, in denen die `new`-Komponenten nicht leer sind, gilt, dass die Funktion `all_subfrmls` bzgl. den jeweiligen `new`-Komponenten der Aufrufe von `expand` konstant bleibt. Der Grund hierfür ist, dass während der Berechnung des aktuellen Zustands (Teil-)Formeln aus der `new`-Komponente in die `old`- und `next`-Felder der `node`-Komponente umgeschichtet werden. Für den Fall der leeren `new`-Komponente hat man im Allgemeinen nur eine Inklusionsbeziehung, da das erste Argument des rekursiven Aufrufs nur die LTL-Formeln enthält, die im `next`-Feld der ursprünglichen `node`-Komponente enthalten sind. Die LTL-Formeln des `old`-Felds werden hier nicht mehr berücksichtigt, fließen aber in das zweite Argument mit ein. Folglich ist es erforderlich bei der Konstruktion der oberen Schranke `LIM` ebenfalls das zweite Argument zu berücksichtigen. Folgende Definition liefert schließlich eine geeignete obere Schranke.

```
definition nds_limit :: "[cnode, node list] ⇒ (frml set × frml set) set"
where
  "nds_limit n ns
   ≡ (Pow (all_subfrmls n ∪ (⋃n'∈set ns. all_subfrmls ([], n'))))
      × (Pow (all_subfrmls n ∪ (⋃n'∈set ns. all_subfrmls ([], n'))))"
```

Die Funktion `nds_limit` ordnet den Argumenten von `expand` eine Menge zu, die alle möglichen Kombinationen von Paaren enthält, deren Komponenten gerade aus Teilformeln der Formeln bestehen, die in `n` und `ns` enthalten sind.

Die durch `nds_limit` berechnete Menge ist während der gesamten Berechnung konstant, d. h. sie liefert das gleiche Ergebnis sowohl auf den Argumenten der linken Seite einer Gleichung als auch auf den rekursiven Argumenten. Ausgehend von den Argumenten `([], n)` und `ns` des Falls der leeren `new`-Komponenten etwa hat man als Argumente des rekursiven Aufrufs von `expand` gerade

$$\texttt{(next n, (|..., old = [], next = []|))}$$

und `n#ns`. Sei `arg` eine Abkürzung für das erste Argument des rekursiven Aufrufs. Wie bereits erwähnt, gilt folgende Inklusionseigenschaft:

$$\texttt{all\_subfrmls arg} \subseteq \texttt{all\_subfrmls ([], n)}$$

Unter Verwendung der Definition von `all_subfrmls` erhält man die Gültigkeit dieser Aussage direkt.

Aus obiger Inklusionseigenschaft ergibt sich nun für die Argumente des rekursiven Aufrufs, dass der folgende Teilausdruck aus „`nds_limit arg (n#ns)`"

$$\texttt{all\_subfrmls arg} \cup (\bigcup \texttt{n'} \in \texttt{set (n\#ns). all\_subfrmls ([], n')})$$

identisch mit dem Ausdruck

$$\texttt{all\_subfrmls n} \cup (\bigcup \texttt{n'} \in \texttt{set ns. all\_subfrmls ([], n')})$$

ist. Und genau diese Eigenschaft führt dazu, dass „`nds_limit ([], n) ns`" zu „`nds_limit arg (n#ns)`" gleich sein muss.

In den restlichen Fällen bleibt das zweite Argument sowie `all_subfrmls` angewandt auf das erste Argument konstant, sodass sich `nds_limit` ebenfalls nicht ändert.

Mit dem folgenden Lemma erhält man schließlich, dass „`nds_limit n ns`" tatsächlich die gesuchte obere Schranke `LIM` sein muss.

```
lemma pair_in_nds_limit:
 "old_next_pair (node n) ∈ nds_limit n ns"
```

Mit der Eigenschaft, dass `nds_limit` auf den Argumenten von `expand` stets konstant ist, und dem bereits vorgestellten Lemma `neg_if_cond_transf` folgt die Gültigkeit folgender Beziehung:

```
nds_limit arg (n#ns) - old_next_pair ‘ set (n#ns)
= nds_limit ([], n) ns - old_next_pair ‘ set (n#ns)
⊂ nds_limit ([], n) ns - old_next_pair ‘ set ns
```

Folglich hat man nun mit `nds_limit` eine Funktion gefunden, die eine obere Schranke für das zweite Argument von `expand` berechnet. Mit Hilfe dieser Funktion ist man nun im Stande, eine Ordnung anzugeben, bzgl. der die rekursiven Argumente von `expand` im Fall der leeren `new`-Komponente gegenüber den ursprünglichen Argumenten echt kleiner sind. Außerdem bleiben die rekursiven Argumente in den anderen Fällen der `expand`-Definition höchstens gleich.

5.2.5.3 Terminierungsordnung

Ausgehend von den Betrachtungen in Abschnitt 5.2.5.1 und 5.2.5.2 lässt sich nun eine Ordnung angeben, bzgl. der die rekursiven Argumente von `expand` gegenüber den ursprünglichen Argumenten in der Definition von `expand` stets kleiner werden, sodass die Terminierung von `expand` gewährleistet ist.

Seien `n` und `ns` die Argumente von `expand`. Das folgende Paar induziert eine Terminierungsordnung für `expand`.

```
(nds_limit n ns - old_next_pair ‘ set ns,
 size_frml_list (new n))
```

Betrachtet man solche Paare lexikographisch bzgl. der $\subset$-Beziehung für endliche Mengen und der $<$-Beziehung für natürliche Zahlen, d. h. (A, k) ist kleiner als (B, l) gdw. $A \subset B \vee A = B \wedge k < l$, so folgt, dass solche Paare im Zuge der Rekursion immer kleiner werden. Schließlich gilt im Fall der leeren `new`-Komponente, dass die erste Komponente des obigen Paares für die Argumente des rekursiven Aufrufs echt kleiner wird. Für die restlichen Fälle hat man entweder Gleichheit oder ebenfalls eine Abnahme dieser Menge. Für den Fall der Gleichheit hat man aber die Eigenschaft, dass die zweite Komponente des obigen Paares für die Argumente der rekursiven Aufrufe kleiner wird, sodass in der Tat solche Paare bzgl. der vorgestellten lexikographischen Ordnung dekrementieren. Aus der Tatsache, dass die $\subset$-Beziehung für endliche Mengen und die $<$-Beziehung für natürliche Zahlen lexikographisch betrachtet eine wohlfundierte Ordnung nach sich ziehen, hat man schließlich die Lösung des ursprünglichen Problems, wie bereits zu Beginn des Abschnitts 5.2.5 erläutert, gefunden.

5.3 LGBA-Transformation

In diesem Abschnitt geht es darum, wie ein LGBA (vgl. Abschnitt 2.3.1) aus dem durch die Funktion `create_graph` konstruierten Graphen erzeugt wird. Die Transformation des Graphen in einen LGBA basiert auf der Idee von Gerth et al. [1, Kap. 3.3]. Sei der Graph gemäß der Definition in Abschnitt 5.2.3.5 durch `(V,`Δ`)` gegeben.

5.3.1 Motivation

Ein LGBA kann nach Abschnitt 2.3.1 als ein Tupel $(\mathcal{A}, \mathcal{D}, \mathcal{L})$ aufgefasst werden. Dabei handelt es sich bei $\mathcal{A}$ um den zum LGBA dazugehörigen GBA, der wiederum als ein Tupel der Form (Q, I, δ, F) dargestellt werden kann.

Wie bereits in Kapitel 3 festgestellt, hat $\mathcal{D}$ die Gestalt 2^{Prop}, während Q identisch `V` und δ gleich Δ gesetzt werden, da beim Übergang zum LGBA sowohl die Menge der Zustände `V` als auch die Transitionsfunktion Δ erhalten bleiben.

Die Markierungsfunktion $\mathcal{L}$, die jedem Zustand eine Menge von Propositionsmengen zuordnet, wird mit Hilfe des `old`-Feldes definiert. Im `old`-Feld eines Zustands q werden u. a. (negierte) Propositionen abgelegt, die unmittelbar in diesem Zustand erfüllt sein müssen bzw. nicht erfüllt sein dürfen. Die Markierungsfunktion $\mathcal{L}$ ordnet diesem Zustand q nur solche Propositionsmengen zu, die mit diesem Zustand kompatibel sind, d. h. $M \in \mathcal{L}(q)$ gdw. $M \vDash p$ für alle Propositionen p und $M \nvDash p'$ für alle negierten Propositionen p', die im `old`-Feld von q enthalten sind.

Die Konstruktion des Graphen legt den Bezeichner `init` in das `incoming`-Feld eines Zustands ab, falls die Verarbeitung der Eingabeformel von diesem Zustand aus initiiert wurde. Folglich erhält man gerade alle Startzustände, indem man sich auf Elemente aus `V` einschränkt, die `init` im `incoming`-Feld beinhalten, d. h. I kann gleich „`set [n'←ns. init ∈ set (incoming n')]`" gesetzt werden.

Tatsächlich ist es so, dass die Graphenkonstruktion aus Abschnitt 5.2 für eine LTL-Formel der Form $\mu \mathsf{U} \psi$ eventuell unendliche Pfade konstruiert, die nicht die Modelle dieser Formel bestimmen. Dies wird nun anhand eines Beispiels erläutert.

Beispiel Sei die Eingabeformel von `create_graph` sinngemäß durch $p \,\mathsf{U}\, q$ für zwei verschiedene Propositionen p und q gegeben.

Abbildung 5.2 zeigt, wie der für die Eingabeformel $p \,\mathsf{U}\, q$ konstruierte Graph sinngemäß aussieht. Die Startzustände, also Zustände in deren `incoming`-Feld der

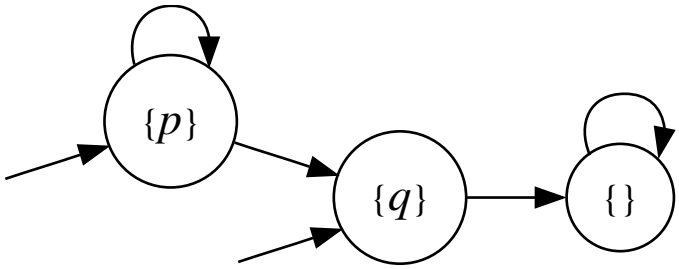

Abbildung 5.2: Graphdarstellung für $p \mathsf{U} q$

Bezeichner `init` enthalten ist, werden durch eine eingehende Kante, die keinen Zustand besitzt, von dem sie ausgeht, symbolisiert. Die Beschriftungen der Zustände sind durch eine Menge gegeben. Diese Menge repräsentiert gerade alle Propositionen, die in den jeweiligen Zuständen erfüllt sein müssen, d. h. es handelt sich dabei um die Propositionen, die in dem jeweiligen `old`-Feld des Zustands abgelegt sind. Man erkennt nun, dass der unendliche Pfad, der mit dem Zustand `{`p`}` beginnt und stets in diesem Zustand verbleibt, offensichtlich keine Modelle für die Eingabeformel liefert, da ein Zustand, in dem die Proposition q schließlich erfüllt ist, so wie es die Semantik der Eingabeformel eigentlich vorsieht, nie auf diesem Pfad erreicht wird.

Um solche Pfade auszuschließen, bietet es sich an, die Akzeptanzbedingung des GBA $\mathcal{A}$ zu verwenden. Diese legt fest, dass nur solche Pfade akzeptiert werden, deren Limit[15] mit den Akzeptanzmengen, gegeben durch F, in mindestens einem Element übereinstimmt. Legt man etwa im obigen Beispiel fest, dass nur der Zustand `{}` ein akzeptierender Zustand ist, so werden nur noch solche unendliche Pfade zugelassen, die tatsächlich die Modelle der Eingabeformel wiedergeben. Das Kernproblem besteht darin, dass das Verfahren ausgehend von der Expansionsstrategie für eine Until-Formel $\xi \vDash \mu \mathsf{U} \psi \equiv \xi \vDash \psi \vee (\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi))$ eine Verzweigung konstruiert. Der eine Pfad dieser Verzweigung sollte die LTL-Formel ψ modellieren, der andere die LTL-Formel $\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi)$. Im letzteren Fall stellt sich das Verfahren im nächsten Zustand erneut der Aufgabe, die Pfade für $\mu \mathsf{U} \psi$ zu konstruieren. Sukzessive fortgesetzt erhält man in der Regel einen unendlichen Pfad, auf dem für jeden Zustand q dieses Pfades gilt: $\mu \mathsf{U} \psi \in \mathtt{old}(q) \wedge \psi \notin \mathtt{old}(q)$[16]. Wie bereits erwähnt, werden im `old`-Feld eines Zustands gerade LTL-Formeln

[15] Das Limit eines Pfades ist die Menge der Zustände, die auf dem Pfad unendlich oft besucht werden (vgl. Abschnitt 2.3.1).

[16] Diese Behauptung lässt sich auch formal nachweisen und wird im Korrektheitsbeweis verwendet (vgl. Lemma 4.2 in Abschnitt 5.3.3.2).

abgelegt, die im Zuge der Verarbeitung betrachtet wurden. Die Zustände, die obige Bedingung erfüllen, sollen nicht akzeptierend sein. Ist das der Fall, so wird der oben beschriebene Pfad durch den GBA $\mathcal{A}$ zurückgewiesen, da das Limit dieses Pfades gerade nur nicht akzeptierende Zustände enthält. Man legt also fest, dass Zustände, die obige Bedingung erfüllen, nicht in den Akzeptanzmengen auftauchen dürfen. Dazu wählt man nur die Zustände q' aus, sodass q' die Negation obiger Bedingung, also $\mu \mathsf{U} \psi \in \mathit{old}(q') \rightarrow \psi \in \mathit{old}(q')$, erfüllt. Ausgehend von der Eingabeformel definiert man nach diesem Prinzip für jede Teilformel der Form $\mu \mathsf{U} \psi$ eine Akzeptanzmenge, um die Gültigkeit aller akzeptierter Pfade zu gewährleisten.

5.3.2 Modellierung des LGBA

Ausgehend von den bisherigen Betrachtungen gilt es nun, eine Funktion zu implementieren, die den LGBA konstruiert. Dazu ist zunächst eine Modellierung des LGBA notwendig. Unter dem Aspekt der Codegenerierung muss diese so beschaffen sein, dass der Codegenerator von Isabelle/HOL in der Lage ist, aus der Definition Code zu erzeugen. Insbesondere bedeutet das, dass abstrakte Strukturen wie Mengen durch konkrete Strukturen wie Listen repräsentiert werden müssen. Da außerdem die Korrektheit der LGBA-Konstruktion nachgewiesen werden soll, ist es erforderlich, neben der Implementierung des LGBA zusätzlich dessen Spezifikation anzugeben. Für diese Spezifikation ist es nicht notwendig, Code zu generieren, sodass diese wiederum so abstrakt wie möglich definiert werden kann.

Der GBA des LGBA hat folgende Gestalt:

```
record 'q gba =
  initial :: "'q list"
  trans   :: "('q × 'q) list"
  accept  :: "'q list list"
```

Der Typparameter `'q` repräsentiert Zustände. Das Feld `initial` modelliert die Startzustände. Wie bereits erwähnt, verwende ich zwecks Codegenerierung an dieser Stelle Listen statt Mengen. Ähnliches gilt für die Transitionsfunktion, die hier durch das Feld `trans` gegeben ist. Diese wird hier durch eine Liste von Paaren dargestellt. Solche Paare werden, wie man später feststellen wird, die Gestalt `(q,Δ(q))` für die durch den Graph induzierte Transitionsfunktion Δ (vgl. Abschnitt 5.2.3.5) haben. Die Akzeptanzmengen, gegeben durch das Feld `accept`, werden ebenfalls durch Listen repräsentiert. Gegenüber der abstrakten Definition

eines LGBA in Abschnitt 2.3.1 fehlt dieser Modellierung die Menge aller Zustände. Da diese jedoch implizit im Feld `trans` verfügbar sein wird, besteht darauf bezogen kein Nachteil gegenüber der abstrakten Definition. Des Weiteren wird die Menge aller Zustände höchstens für die Ausgabe des LGBA benötigt. Für diesen Zweck ist aber die Möglichkeit der Rekonstruktion dieser Menge aus den Transitionen gegeben.

Die Repräsentation des LGBA sieht wie folgt aus.

```
record 'q lgba =
  gbauto :: "'q gba"
  label  :: "'q ⇀ prop list list"
```

Der GBA ist durch das Feld `gbauto` gegeben. Die Markierungsfunktion wird durch eine Map-Struktur[17] implementiert, die nur auf den Zuständen des GBA definiert ist. Dabei ordnet die Markierungsfunktion einem Zustand eine Liste von `prop`-Listen zu. Der Typ `prop` repräsentiert Propositionen, so wie das in Abschnitt 5.1 beschrieben ist. Die Menge $\mathcal{D}$, die gerade identisch zu 2^{Prop} sein muss, wird von der Implementierung nur implizit festgelegt.

5.3.2.1 Konstruktion des GBA

Bevor der LGBA erzeugt werden kann, ist die Konstruktion des dazugehörigen GBA erforderlich. Wie bereits erwähnt, werden die Startzustände und Transitionen des GBA aus dem durch `create_graph` induzierten Graphen, so wie in Abschnitt 5.2.3.5 beschrieben, konstruiert. Die Akzeptanzmengen, wie zu Beginn des Abschnitts 5.3 erläutert, werden dabei dadurch bestimmt, indem man für jede Teilformel der Eingabeformel der Form $\mu \mathsf{U} \psi$ eine Menge konstruiert, die nur solche Zustände enthält, die nur zulässige unendliche Pfade zulassen. Dazu ist es zunächst erforderlich, ausgehend von der Eingabeformel alle Until-Formeln zu bestimmen.

```
fun all_Until_frmls :: "frml ⇒ frml list"
where
  "all_Until_frmls (φ and_n ψ)
   = all_Until_frmls φ @ all_Until_frmls ψ"
```

[17] Eine Map, repräsentiert durch ⇀, modelliert eine partielle Funktion. Das Map-Konzept ähnelt dem gleichnamigen Konzept funktionaler Programmiersprachen.

```
| "all_Until_frmls (φ or_n ψ)
   = all_Until_frmls φ @ all_Until_frmls ψ"
| "all_Until_frmls (X_n φ) = all_Until_frmls φ"
| "all_Until_frmls (φ V_n ψ)
   = all_Until_frmls φ @ all_Until_frmls ψ"
| "all_Until_frmls (φ U_n ψ)
   = φ U_n ψ # all_Until_frmls φ @ all_Until_frmls ψ"
| "all_Until_frmls _ = []"
```

Die Funktion `all_Until_frmls` bestimmt für eine LTL-Formel rekursiv alle Teilformeln der Form $\phi \mathsf{U} \psi$ und gibt diese als Ergebnis zurück. Die Akzeptanzmengen werden auf Basis dieser Funktion konstruiert. Dies geschieht wie folgt:

```
definition accept_cond :: "[frml, node list] ⇒ node list list"
where
  "accept_cond φ ns
   ≡ map (λf. case f of
               _ U_n ψ ⇒ [q←ns. f∈set (old q) ⟶ ψ∈set (old q)])
         (remdups (all_Until_frmls φ))"
```

Mit Hilfe der Funktion `accept_cond`, die als Parameter die Eingabeformel sowie die Liste der durch `create_graph` erzeugten Zustände erhält, wird jeder Until-Formel, die eine Teilformel der Eingabeformel ist, eine Liste von Zuständen, die genau die Bedingung erfüllt, wie sie zu Beginn des Abschnitts 5.3 erläutert wurde, zugeordnet. Da die Hilfsfunktion `all_Until_frmls` ggf. mehrere gleiche Elemente produziert, werden doppelte Elemente mit der Funktion `remdups` eliminiert.

Folgende Funktion definiert schließlich den GBA des LGBA:

```
definition create_gba :: "[frml, node list] ⇒ node gba"
where
  "create_gba φ ns
   ≡ (| initial = [q←ns. init ∈ set (incoming q)],
        trans = concat (map (λq. (map (λp. (q, p))
                                      [p←ns. name q ∈ set (incoming p)]))
                            ns),
        accept = accept_cond φ ns |)"
```

Der Funktion `create_gba` werden als Parameter die Eingabeformel und die Liste der Zustände übergeben. Daraus werden Startzustände, d. h. Zustände, die mit `init` markiert sind, extrahiert. Ebenfalls werden Transitionen bestimmt, also Paare `(q,p)`, sodass „`name q ∈ set (incoming p)`" erfüllt ist. Die Akzeptanzmengen werden mit der gerade definierten Hilfsfunktion `accept_cond` berechnet.

5.3.2.2 Konstruktion des LGBA

Der LGBA wird ausgehend von der Konstruktion des GBA aus Abschnitt 5.3.2.1 konstruiert. Zusätzlich zum GBA ist die Konstruktion der Markierungsfunktion erforderlich. Die Markierungsfunktion ordnet jedem Zustand die Markierungen zu, die kompatibel mit dem Zustand sind, d. h. die alle positiven und keine negative Propositionen, die in dem Zustand erfüllt sein müssen bzw. nicht erfüllt sein dürfen, enthalten. Problematisch ist aber die Tatsache, dass die Menge Prop gemäß der Definition in Abschnitt 2.3.1 endlich sein muss. Bisher wurden Propositionen durch den Typen `prop` repräsentiert. Diese Vorgehensweise erlaubt es, beliebig viele Propositionen zu modellieren, bedingt jedoch eine unendliche Menge von Propositionen. Ist diese Menge aber unendlich, so sind ebenfalls unendlich viele Markierungen vorhanden. Letztendlich werden Propositionen jedoch vom Benutzer, der die Spezifikation in Form einer LTL-Formel vorgibt, bestimmt. Folglich müsste neben der Eingabeformel eine endliche Menge von Propositionen zur Eingabe zählen, die bestimmt, welche Propositionen in der Eingabeformel auftauchen können. Alternativ würde es auch ausreichen, aus der Eingabeformel die Propositionen zu extrahieren, die darin enthalten sind. Auf diese Weise würde man eine endliche Menge von Propositionen erhalten, die mit der Eingabeformel konsistent (d. h. alle in der Eingabeformel enthaltene Propositionen sind in dieser endlichen Menge enthalten) ist. Genau diesen Ansatz verfolge ich in meiner Implementierung. Für diese Zwecke gibt es eine Funktion, die von einer LTL-Formel ausgehend alle darin enthaltene Propositionen bestimmt.

```
fun get_props :: "frml ⇒ prop list"
where
  "get_props prop_n(p) = [p]"
| "get_props nprop_n(p) = [p]"
| "get_props (φ and_n ψ) = get_props φ @ get_props ψ"
| "get_props (φ or_n ψ) = get_props φ @ get_props ψ"
| "get_props (X_n φ) = get_props φ"
| "get_props (φ V_n ψ) = get_props φ @ get_props ψ"
| "get_props (φ U_n ψ) = get_props φ @ get_props ψ"
| "get_props _ = []"
```

Die Funktion `get_props` bestimmt rekursiv alle in der Eingabeformel enthaltene Propositionen. Da die Ausgabe eine Liste darstellt, können einige Propositionen doppelt auftauchen.

Die positiven und negativen Propositionen eines Zustands können aus dem `old`-Feld bestimmt werden. Um diese zu extrahieren, existieren folgende Funktionen.

```
fun pos_props :: "frml list ⇒ prop list"
where
  "pos_props [] = []"
| "pos_props (prop_n(p)#xs) = p # (pos_props xs)"
| "pos_props (_#xs) = pos_props xs"

fun neg_props :: "frml list ⇒ prop list"
where
  "neg_props [] = []"
| "neg_props (nprop_n(p)#xs) = p # (neg_props xs)"
| "neg_props (_#xs) = neg_props xs"
```

Zwar wäre auch eine Implementierung einer Funktion möglich, die beide Ergebnisse gleichzeitig liefert. Der besseren Übersicht halber wird hier darauf verzichtet.

Die Kernfunktionalität der Markierungsfunktion liefert folgende Definition.

```
definition label_fct :: "[prop list list, node] ⇒ prop list list"
where
  "label_fct Lbls n
   ≡ [xs←Lbls. set (pos_props (old n)) ⊆ set xs
               ∧ list_inter xs (neg_props (old n)) = []]"
```

Dabei repräsentiert der Parameter `Lbls` gerade alle möglichen Markierungen als Listen ausgehend von den Propositionen, die in der Eingabeformel enthalten sind, jedoch mit einer Einschränkung: Für eine Reihe von Propositionen existiert genau eine Liste in `Lbls`, die eine Permutation dieser Reihe darstellt, d. h. für zwei verschiedene Propositionen `p` und `q` ist entweder die Liste `[p,q]` oder die Liste `[q,p]` in `Lbls` enthalten. Für Markierungen ist die Reihenfolge der Elemente völlig irrelevant, da die Semantik auf LTL-Formeln letztendlich nur das Enthaltensein oder das Nichtenthaltensein einer Proposition in einer Markierung fordert. Die Funktion `label_fct` berechnet die für einen Zustand kompatiblen Markierungen, d. h. solche Markierungen, die mindestens die positiven Propositionen des Zustands enthalten, negative Propositionen jedoch nicht. Die Hilfsfunktion `list_inter` berechnet für zwei Listen `ys` und `zs` eine Liste, die die gemeinsamen Elemente von `ys` und `zs` enthält.

Nun kann man die Konstruktion des LGBA wie folgt angeben.

```
definition create_lgba :: "frml ⇒ node lgba"
where
  "create_lgba φ
   ≡ (let ns = create_graph φ in
       ⦇ gbauto = create_gba φ ns,
         label = [ns[↦]map (label_fct (list_Pow (get_props φ))) ns] ⦈)"
```

Ausgehend von der Eingabeformel φ wird zunächst der Graph mittels der Funktion `create_graph` berechnet. Wie bereits in Abschnitt 5.3.2.1 erläutert, wird aus diesem Graphen der GBA konstruiert. Die Markierungsfunktion wird als Map so festgelegt, dass sie jedem Zustand eine Liste von Markierungen zuordnet. Diese Markierungen können nur Propositionen enthalten, die in der Eingabeformel auftauchen. Mittels `list_Pow` erhält man, ähnlich wie bei einer Potenzmengenkonstruktion, alle mögliche Markierungen basierend auf den Propositionen der Eingabeformel. Da dabei die Reihenfolge der Elemente vernachlässigt wird, liefert `list_Pow` für eine Reihe von Propositionen genau eine Liste, die diese Propositionen enthält.

5.3.3 Beweis der Korrektheit

In diesem Abschnitt geht es nun darum, zu klären, ob der in Abschnitt 5.3.2 durch die Funktion `create_lgba` definierte LGBA sich korrekt verhält, d. h. es stellt sich die Frage, ob dieser LGBA für die Eingabeformel ϕ exakt die ω-Wörter über 2^{Prop} akzeptiert[18], die die LTL-Formel ϕ erfüllen. Um diese Spezifikation des LGBA zu formulieren, sind entsprechende Definitionen notwendig.

5.3.3.1 Formalisierung der LGBA-Spezifikation

Zunächst sind folgende Begriffe erforderlich, um festzulegen, wann eine unendliche Folge von Zuständen als Pfad eines GBA bezeichnet wird.

```
abbreviation gba_run :: "['q gba, 'q word] ⇒ bool"
where
  "gba_run A σ
   ≡ ∀n. ((σ n), σ (Suc n)) ∈ set (trans A)"
```

[18] Die Menge Prop wird stets als die Menge der Propositionen aufgefasst, die in der Eingabeformel ϕ enthalten sind.

```
definition gba_exec :: "['q gba, 'q word] ⇒ bool"
where
  "gba_exec A σ
   ≡ σ 0 ∈ set (initial A) ∧ gba_run A σ"
```

Diese Definition orientiert sich an der abstrakten Definition in Abschnitt 2.3.1. Dabei fordert das Prädikat[19] `gba_run` für einen GBA `A` und ein ω-Wort σ über der Menge der Zustände von `A` zunächst, dass es sich bei σ nur um einen Teilpfad von `A`, der nicht zwingenderweise in einem Startzustand beginnt, handelt. Während das Prädikat `gba_exec` basierend darauf einen Pfad gemäß Abschnitt 2.3.1 beschreibt.

Die Akzeptanzbedingung des GBA kann wie folgt formuliert werden:

```
abbreviation gba_accept_cond :: "['q gba, 'q word] ⇒ bool"
where
  "gba_accept_cond A σ
   ≡ ∀i<length (accept A). limit σ ∩ set (accept A!i) ≠ {}"

definition gba_accept :: "['q gba, 'q word] ⇒ bool"
where
  "gba_accept A σ
   ≡ gba_exec A σ ∧ gba_accept_cond A σ"
```

Das Limit eines ω-Wortes w entspricht gemäß Abschnitt 2.3.1 der Menge von Elementen von w, die in w unendlich oft vorkommen. Genau diese Menge berechnet die Funktion `limit`.

Die Akzeptanzbedingung des LGBA wird wie folgt definiert:

```
abbreviation lgba_accept_cond
            :: "['q lgba, 'q word, (prop set) word] ⇒ bool"
where
  "lgba_accept_cond A σ ξ
   ≡ ∀i. ∃W. label A (σ i) = Some W
             ∧ (∃w∈set W. ξ i = set w)"
```

[19] Eine Funktion, deren Rückgabewert den Typ `bool` hat, wird als Prädikat bezeichnet.

```
definition lgba_accept :: "['q lgba, (prop set) word] ⇒ bool"
where
  "lgba_accept A ξ
   ≡ ∃σ. gba_accept (gbauto A) σ ∧ lgba_accept_cond A σ ξ"
```

Die Notation in Hinblick auf die Akzeptanz des ω-Wortes ξ sieht auf den ersten Blick recht umständlich aus. Bedingt wird das durch die Repräsentation der Markierungsfunktion. Im Wesentlichen soll diese Bedingung aussagen, dass $\xi(i) \in \mathcal{L}(\sigma(i))$ für $i \in \mathbb{N}$ erfüllt ist. Da die Markierungsfunktion in der Implementierung durch Maps modelliert wird, erhält man den Rückgabewert der Markierungsfunktion zunächst durch „`label A (σ i) = Some W`"[20]. Dabei enthält `W` eine Liste von Propositionslisten, die gerade alle Markierungen des Zustands „σ `i`" repräsentieren. Diese Markierungen, die aus Propositionen bestehen, haben aber bedingt durch die Darstellung als Listen eine Reihenfolge der Elemente. Um die Reihenfolge der Elemente zu eliminieren, reicht es aus, die Liste mittels `set` in eine Menge zu überführen.

5.3.3.2 Theorem 4.1

Mit der Formalisierung der Akzeptanzbedingung eines LGBA aus dem vorhergehenden Abschnitt 5.3.3.1 lässt sich nun die Korrektheitseigenschaft wie folgt formulieren:

```
theorem T4_1:
  assumes "∀i. ξ i ∈ Pow (set (get_props φ))"
    shows "lgba_accept (create_lgba φ) ξ ⟷ ξ ⊨n φ"
```

Die Voraussetzung dieses Theorems fordert zunächst im Wesentlichen, dass ξ ein ω-Wort über 2^{Prop} sein muss, wobei die Menge Prop als die Menge der Propositionen, die in der LTL-Formel φ auftauchen, aufgefasst wird. Dabei extrahiert die Funktion `get_props` gerade aus der Eingabeformel φ die darin enthaltenen Propositionen, wie in Abschnitt 5.3.2.2 beschrieben. Es würde auch wenig Sinn machen, beliebige endliche Mengen Prop zuzulassen. Das würde dazu führen, dass man gezwungen wäre, für eine LTL-Formel ϕ und Prop eine Wohlgeformtheitseigenschaft zu fordern, um zu gewährleisten, dass die Propositionen in ϕ tatsächlich

[20] Für eine Map `m` bedeutet `m x = Some a`, dass `m` auf dem Wert `x` *definiert* ist und den Wert `a` liefert.

auch in Prop enthalten sind. Außerdem tragen Propositionen, die nicht in ϕ enthalten sind, gemäß der Definition der Semantik (vgl. Abschnitt 2.2) nichts zur Erfüllbarkeit von ϕ bei. Schließlich besagt Theorem 4.1, dass unter der oben beschriebenen Voraussetzung das ω-Wort ξ über 2^{Prop} die LTL-Formel in Negationsnormalform φ genau dann erfüllt[21], wenn der aus φ durch die Funktion `create_lgba` konstruierte LGBA ξ akzeptiert.

Der Beweis des Theorems 4.1 basiert auf der Beweisidee von Gerth et al. [1, Kap. 4]. Die im Folgenden vorgenommene Nummerierung der Aussagen ist im Großen und Ganzen ebenfalls an die ursprüngliche Beweisidee angelehnt. Die zwei Richtungen des Theorems werden separat in Lemma 4.9 (Seite 69) und Lemma 4.10 (Seite 72) gezeigt. Während die Implikation von links nach rechts (Lemma 4.9) die Voraussetzung nicht zwingend benötigt, ist sie in der umgekehrten Richtung (Lemma 4.10) zwingend erforderlich. Doch bevor beide Richtungen gezeigt werden können, ist es zunächst notwendig, sich mit einem Problem der LGBA-Konstruktion (vgl. Abschnitt 5.3) zu beschäftigen, das darin besteht, dass die Graphenkonstruktion aus Abschnitt 5.2 ggf. unendliche Pfade konstruiert, die unzulässige Modelle für die Eingabeformel implizieren. Das nachfolgende Lemma verdeutlicht das Problem.

Lemma 4.2 Die Grundidee der LGBA-Konstruktion besteht darin, in die `old`-Felder eines Zustands die LTL-Formeln abzulegen, für die der nachfolgend konstruierte Teilpfad entsprechende Modelle dieser Formeln impliziert. Nun werden aber ggf. Teilpfade konstruiert, die dieser Idee widersprechen, wie das folgende Lemma zeigt:

```
lemma L4_2a:
  assumes "gba_run (gbauto (create_lgba φ)) σ"
      and "f Un g ∈ set (old (σ 0))"
    shows "(∀i. {f, f Un g} ⊆ set (old (σ i))
                ∧ g ∉ set (old (σ i)))
           ∨ (∃j. (∀i<j. {f, f Un g} ⊆ set (old (σ i)))
                  ∧ g ∈ set (old (σ j)))"
```

Demnach gilt für einen Teilpfad σ des LGBA, der im eigentlichen Sinne u. a. Modelle für die Until-Formel der Form $f \,\mathsf{U}\, g$ implizieren soll, dass auf diesem Teilpfad ein Folgezustand existiert, von dem aus Modelle für g impliziert werden,

[21] Die Relation $\models_n$ ist für LTL-Formeln in Negationsnormalform analog zur Relation $\models$ für allgemeine LTL-Formeln, wie in Abschnitt 5.1.2 beschrieben, definiert.

oder dass es keinen solchen Zustand gibt. Im letzteren Fall würde der LGBA Teilpfade zulassen, die gerade nicht die Modelle der Eingabeformel bestimmen. Um solche unzulässige Teilpfade zurück zuweisen, verwendet man die Akzeptanzbedingung des zum LGBA zugehörigen GBA, wie das folgende Lemma zeigt.

lemma `L4_2b`:
  assumes "`gba_run (gbauto (create_lgba` φ`)) `σ"
    and "`f U`$_n$ `g ∈ set (old (`σ `0))`"
    and "`gba_accept_cond (gbauto (create_lgba` φ`)) `σ"
  shows "$\exists$`j. (∀i<j. {f, f U`$_n$ `g} ⊆ set (old (`σ `i)))`
        `∧ g ∈ set (old (`σ `j))`"

Verwendet man also zusätzlich zu den Voraussetzungen aus `L4_2a` die Akzeptanzbedingung des GBA, so werden unzulässige Teilpfade nicht mehr zugelassen.

Die Beweisidee für Lemma `L4_2a`, und damit auch für Lemma `L4_2b`, basiert auf der Tatsache, dass die Disjunktionsglieder der Aussage von `L4_2a` exklusiv sind, d. h. wenn das eine Disjunktionsglied erfüllt ist, dann ist das andere nicht erfüllt, und umgekehrt. Dazu zeigt man etwa, dass aus der Negation des rechten Disjunktionsglieds das linke Disjunktionsglied folgt. Lemma `L4_2b` folgt dann direkt aus Lemma `L4_2a` durch einen Widerspruchsbeweis. Nimmt man dazu die Negation der Aussage von `L4_2b` an, so lässt sich mit `L4_2a` die Aussage folgern, die besagt, dass es sich um einen unzulässigen Teilpfad handelt. Man erhält also folgende Aussage:

$$\forall \texttt{i. \{f, f U}_n \texttt{ g\}} \subseteq \texttt{set (old (}\sigma \texttt{ i))} \wedge \texttt{g} \notin \texttt{set (old (}\sigma \texttt{ i))}$$

Da die Until-Formel `f U`$_n$ `g` eine Teilformel der Eingabeformel φ sein muss[22], existiert gemäß der Definition der Akzeptanzbedingung des GBA (vgl. Abschnitt 5.3.2.1) eine zu der Until-Formel zugehörige Akzeptanzmenge M. Diese Menge M enthält dabei nur solche Zustände `q`, die folgende Eigenschaft erfüllen:

$$\texttt{f U}_n \texttt{ g}\in\texttt{set (old q)} \longrightarrow \texttt{g}\in\texttt{set (old q)}$$

Zusätzlich gilt, dass das Limit von σ mit der Menge M mindestens ein gemeinsames Element besitzt. Sei ein solches Element gerade durch `q` gegeben. Da `q` zum Limit von σ gehört, muss `q` in σ enthalten sein. Aus der Unzulässigkeit

[22] In die `old`-Felder von Zuständen werden durch die Graphenkonstruktion aus Abschnitt 5.2 stets nur Teilformeln der Eingabeformel abgelegt.

von σ („`g`$\notin$`set (old q)`") und der Beschaffenheit von `q` („`g`$\in$`set (old q)`") folgt schließlich der Widerspruch, der die Beweisidee abschließt.

Im Gegensatz zu einer Until-Formel ergeben sich aus der Konstruktion der Pfade für eine Release-Formel keinerlei Probleme. Hier lässt sich auch ohne Akzeptanzbedingung des GBA ein entsprechender Bezug zwischen den `old`-Feldern und der Semantik der Release-Formel, wie das nachfolgende Lemma zeigt, herstellen.

```
lemma L4_2c:
  assumes "gba_run (gbauto (create_lgba φ)) σ"
      and "f V_n g ∈ set (old (σ 0))"
    shows "∀i. g ∈ set (old (σ i))
               ∨ (∃j<i. f ∈ set (old (σ j)))"
```

Um diese Aussage zu zeigen, kann man Induktion über `i` verwenden. Zusätzlich ist die Eigenschaft nützlich, die besagt, dass `g` und „`f V`$_n$ `g`" in jedem Fall solange in den `old`-Feldern der Zustände eines Pfades verbleiben, bis irgendwann nicht notwendigerweise ein Zustand erreicht wird, in dem `f` im `old`-Feld enthalten ist.

5.3.3.3 Beweis der Hinrichtung

Um die Implikation des Theorems 4.1 aus Abschnitt 5.3.3.2 von links nach rechts zu zeigen, sind zwei essentielle Aussagen erforderlich. Eine dieser Aussagen (vgl. Lemma 4.6 auf Seite 68) besagt, dass im `old`-Feld eines Startzustands stets die Eingabeformel enthalten ist, die andere (vgl. Lemma 4.8 auf Seite 69) besagt im Wesentlichen, dass eine LTL-Formel erfüllbar ist, falls sie zu Beginn eines Pfades im `old`-Feld vorliegt. Diese zwei Aussagen werden schließlich dazu verwendet um die Hinrichtung von Theorem 4.1 (vgl. Lemma 4.9) zu beweisen.

Lemma 4.6 Wie das nachfolgende Lemma zeigt, gewährleistet die Graphenkonstruktion aus Abschnitt 5.2 die Eigenschaft, dass in dem `old`-Feld eines Startzustands stets die Eingabeformel enthalten ist.

```
lemma L4_6:
  assumes "q ∈ set (initial (gbauto (create_lgba φ)))"
    shows "φ ∈ set (old q)"
```

Der Beweis dieser Aussage beschränkt sich hauptsächlich darauf, zu zeigen, dass für einen Zustand `q` des LGBA, für den „`init ∈ set (incoming q)`" erfüllt ist, die Eigenschaft „φ `∈ set (old q)`" für die Eingabeformel φ gilt. Diese Aussage folgt aber direkt aus der Konstruktion des Graphen (vgl. Abschnitt 5.2).

Lemma 4.8 Basierend auf den bisherigen Betrachtungen (vgl. Lemma 4.2 in Abschnitt 5.3.3.2) lässt sich nur der Zusammenhang zwischen Erfüllbarkeit einer LTL-Formel und dem `old`-Feld eines Zustands herstellen.

```
lemma L4_8:
  assumes "gba_accept (gbauto (create_lgba φ)) σ"
      and "lgba_accept_cond (create_lgba φ) σ ξ"
      and "f ∈ set (old (σ 0))"
    shows "ξ ⊨n f"
```

Demnach gilt, dass, wenn für einen Pfad σ die Akzeptanzbedingung des GBA sowie für σ und ein ω-Wort über 2^{Prop} ξ die Akzeptanzbedingung des LGBA erfüllt ist, dann ist ξ ein Modell für eine LTL-Formel, die sich im `old`-Feld des Startzustandes von σ[23] befindet. Die Aussage dieses Lemmas folgt im Wesentlichen direkt aus der Konstruktion des Graphen (vgl. Abschnitt 5.2) per Induktion über die Struktur von `f` und unter Berücksichtigung von Lemma 4.2 (vgl. Abschnitt 5.3.3.2).

Lemma 4.9 Mit Lemma 4.6 und Lemma 4.8 folgt nun die Hinrichtung des Theorems 4.1 (vgl. Abschnitt 5.3.3.2), die wie folgt lautet:

```
lemma L4_9:
  assumes "lgba_accept (create_lgba φ) ξ"
    shows "ξ ⊨n φ"
```

Zunächst folgt aus der Voraussetzung dieser Aussage gemäß der Definition von `lgba_accept` aus Abschnitt 5.3.3.1 die Existenz eines Pfades σ, der vom GBA und vom LGBA bzgl. ξ akzeptiert wird. Außerdem folgt aus Lemma 4.6, dass die Eingabeformel φ, im `old`-Feld des Startzustands von σ enthalten ist. Mit Lemma 4.8 erhält man schließlich die Aussage, dass ξ ein Modell für φ sein muss.

[23] Der Startzustand eines Pfades σ ist der Zustand, von dem aus der Pfad beginnt, also $\sigma(0)$.

5.3.3.4 Beweis der Rückrichtung

Der Beweis der Implikation des Theorems 4.1 aus Abschnitt 5.3.3.2 von rechts nach links erfordert einen konstruktiven Ansatz. Demnach ist es notwendig, die Existenz eines Pfades nachzuweisen, der die Modelle der Eingabeformel akzeptiert. Dazu reicht es, einen solchen Pfad anzugeben. Entscheidend für der Konstruktion dieses Pfades ist erneut die Problematik, mit der sich Lemma 4.2 (vgl. Abschnitt 5.3.3.2) auseinander setzt. Folglich muss der konstruierte Pfad in jedem Fall die Akzeptanzmengen so berücksichtigen, dass der Pfad die Modelle der Eingabeformel tatsächlich erfüllt.

Zunächst ist ein Begriff erforderlich, der die Konjunktion von LTL-Formeln auf eine Liste von LTL-Formeln verallgemeinert:

abbreviation `list_and_n :: "frml list ⇒ frml" ("⋀n _" [80] 80)`
where
`"list_and_n fs ≡ foldr (λg f. (f and`$_n$` g)) fs true`$_n$`"`

Ausgehend von einer Liste von LTL-Formeln `fs` liefert der Ausdruck „$\bigwedge_n$ `fs`" eine LTL-Formel, die die Elemente der Liste `fs` konjunktiv verknüpft. Mit dieser Notation lässt sich nun eine wichtige Eigenschaft herleiten, die für die Konstruktion des oben beschriebenen Pfades erforderlich ist.

Das Verfahren zur Konstruktion des Graphen basiert auf einer Expansionsstrategie, die die Eingabeformel rekursiv in die Teilformeln zerlegt, die unmittelbar erfüllt sind und die im nächsten Zustand erfüllt sein müssen. Diese Teilformeln werden jeweils in das `old`- und in das `next`-Feld des jeweiligen Zustands abgelegt. Geht man nun davon aus, dass die Eingabeformel φ im `old`-Feld eines Zustands `q` enthalten ist und dass für ein ω-Wort ξ über 2^{Prop} die Eigenschaft „$\xi \models_n \varphi$" erfüllt, so muss gemäß der Expansionsstrategie die Aussage „$\xi \models_n$ `(`$\bigwedge_n$ `old q) and`$_n$ `X`$_n$ `(`$\bigwedge_n$ `next q)`"[24] ebenfalls erfüllt sein, da das `old`-Feld von `q` neben φ Teilformeln von φ und das `next`-Feld von `q`, die Teilformeln von φ, die im nächsten Zustand erfüllt sein müssen, enthält. Um nun den zu konstruierenden Pfad ausgehend vom Zustand `q` weiter fortsetzen zu können, ist zunächst die Existenz eines Folgezustands erforderlich, der geeignet gewählt werden muss. Wie diese Wahl des Folgezustands von `q` vorgenommen werden kann, beschreibt das nachfolgende Lemma.

[24] Diese Aussage wird als ξ ist ein Modell für `q` abgekürzt.

Lemma 4.5 Dieses Lemma zeigt, dass für einen bestimmten Zustand ein geeigneter Folgezustand existiert, sodass dieser für die Konstruktion des Pfades, der für die Rückrichtung des Theorems 4.1 erforderlich ist, verwendet werden kann.

```
lemma L4_5:
  assumes "q∈set (create_graph φ)"
      and "ξ ⊨n (⋀n old q) andn Xn (⋀n next q)"
    shows "∃q'∈set (create_graph φ).
             name q ∈ set (incoming q') ∧
             suffix 1 ξ ⊨n (⋀n old q') andn Xn (⋀n next q') ∧
             {η. ∃μ. μ Un η ∈ set (old q')
                     ∧ suffix 1 ξ ⊨n η} ⊆ set (old q')"
```

Ausgehend von einem Zustand `q`, für den ξ ein Modell ist, gilt demnach, dass stets ein Folgezustand `q'` existiert[25]. Zusätzlich erfüllt dieser Folgezustand die Eigenschaft, dass „`suffix 1` ξ" ein Modell für `q'` ist, sodass zumindest beim Zustandsübergang die Modelleigenschaft von ξ erhalten bleibt. Die letzte Eigenschaft, die Lemma 4.5 liefert, verhindert, dass der Zustandsübergang nach `q'` unzulässig im Sinne von Lemma 4.2 (vgl. Abschnitt 5.3.3.2) gewählt wird, d. h. diese Wahl gewährleistet, dass auf dem konstruierten Pfad für eine Until-Formel der Form $\mu \mathsf{U} \psi$ stets ein Zustand erreicht wird, von dem aus die Modelle für ψ impliziert werden. Da nun der Zustand `q'` im gewissen Sinne (statt ξ verwendet man „`suffix 1` ξ") ebenfalls die Voraussetzungen von Lemma 4.5 erfüllt, kann damit sukzessive eine unendliche Folge von Zuständen, so wie sie benötigt wird, konstruiert werden. Jedoch fehlt bisher die Existenz eines Startzustands, von dem aus der entsprechende Pfad beginnt.

Lemma 4.7 Dieses Lemma liefert die Existenz eines Startzustands, der für die Konstruktion des Pfades, der für die Rückrichtung des Theorems 4.1 erforderlich ist, in Betracht gezogen werden kann.

```
lemma L4_7:
  assumes "ξ ⊨n φ"
    shows "∃q∈set (create_graph φ).
             init∈set (incoming q) ∧
             ξ ⊨n (⋀n old q) andn Xn (⋀n next q) ∧
```

[25] Die Eigenschaft „`name q ∈ set (incoming q')`" besagt im ursprünglichen Sinne, dass der Zustand `q'` eine von `q` eingehende Kante besitzt. Umgekehrt betrachtet gilt, dass eine Kante von `q` nach `q'` existiert.

```
                {η. ∃μ. μ Uₙ η ∈ set (old q)
                        ∧ ξ ⊨ₙ η} ⊆ set (old q)"
```

Die Aussage dieses Lemmas besteht darin, dass unter der Annahme, dass ξ ein Modell für die Eingabeformel φ ist, es einen Startzustand gibt, für den ξ ebenfalls ein Modell ist und der gleichzeitig analog zu Lemma 4.5 die Eigenschaft besitzt, zulässig im Sinne von Lemma 4.2 (vgl. Abschnitt 5.3.3.2) zu sein.

Mit den beiden Lemmata 4.5 und 4.7 besteht nun die Möglichkeit einen entsprechend beschaffenen Pfad zu definieren. Dazu ist zunächst mal folgende Funktion hilfreich:

```
fun auto_run :: "[interprt, node list] ⇒ node word"
where
  "auto_run ξ nds 0
   = hd [q←nds. init ∈ set (incoming q) ∧
                ξ ⊨ₙ (⋀ₙ old q) andₙ Xₙ (⋀ₙ next q) ∧
                {η. ∃μ. μ Uₙ η ∈ set (old q) ∧ ξ ⊨ₙ η} ⊆ set (old q)]"
| "auto_run ξ nds (Suc k)
   = hd [q←nds. name (auto_run ξ nds k) ∈ set (incoming q) ∧
                suffix (Suc k) ξ ⊨ₙ (⋀ₙ old q) andₙ Xₙ (⋀ₙ next q) ∧
                {η. ∃μ. μ Uₙ η ∈ set (old q)
                        ∧ suffix (Suc k) ξ ⊨ₙ η} ⊆ set (old q)]"
```

Für geeignet gewählte Parameter ξ und `nds` verhält sich die so definierte Funktion „`auto_run ξ nds`" gerade so, wie es für einen Pfad von Lemma 4.5 und Lemma 4.7 beabsichtigt ist. In der Tat kann „`auto_run ξ nds`" als ω-Wort über der Menge der Knoten (d. h. als „`node word`") aufgefasst werden, da der Typ „`'a word`" in Abschnitt 5.1.2 nur als eine Abkürzung für „`nat ⇒ 'a`" eingeführt wurde. Weiterhin ist die Hilfsfunktion `hd` so definiert, dass sie für eine nicht leere Liste das oberste Element der Liste zurück gibt. Aus obigen Betrachtungen ist es jedoch garantiert, dass unter entsprechenden Voraussetzungen die Listen, die in der Definition von `auto_run` der Funktion `hd` als Parameter übergeben werden, stets nicht leer sind.

Lemma 4.10 Ausgehend von Lemma 4.5, Lemma 4.7 und obiger Hilfsfunktion `auto_run` lässt sich nun die Rückrichtung des Theorems 4.1 herleiten.

```
lemma L4_10:
  assumes "ξ ⊨ₙ φ"
      and "∀i. ξ i ∈ Pow (set (get_props φ))"
    shows "lgba_accept (create_lgba φ) ξ"
```

Dazu betrachtet man zunächst den gesuchten Pfad als das ω-Wort über der Menge der Zustände „`auto_run` ξ `(create_graph` φ`)`" und zeigt dann, dass es sich tatsächlich um den gesuchten Pfad handelt, indem man dafür die Akzeptanzbedingungen des GBA und die des LGBA nachweist. Für die letztere Eigenschaft ist außerdem die zweite Voraussetzung dieses Lemmas notwendig. Weiterhin ist Lemma 4.2 erforderlich, um zu beweisen, dass sich der durch die Konstruktion resultierende Pfad zulässig verhält.

5.3.3.5 Vergleich der Beweisführung

Obwohl die grundlegende Idee der Verifizierung des Theorems 4.1 (vgl. Abschnitt 5.3.3.2) mit der Beweisidee von Gerth et al. [1, Kap. 4] im Wesentlichen übereinstimmt, unterscheiden sich beide Ansätze dennoch im Detail. Der Hauptunterschied besteht in der Beweisführung von Lemma 4.5 (vgl. Abschnitt 5.3.3.4) und Lemma 4.7 (vgl. Abschnitt 5.3.3.4). Während ich diese Lemmata hauptsächlich direkt aus der Konstruktion ableite, definieren Gerth et al. hierzu eine Relation auf Zuständen, die mit Hilfe des *Father*-Feldes (vgl. Abschnitt 3.1.2) realisiert wird. Diese Relation setzt Zustände zu den Zuständen in Beziehung, die auf dem vorhergehenden Pfad eine Verzweigung einleiten. Damit sind Gerth et al. in der Lage die Tatsache zu formulieren, dass mindestens ein Pfad dieser Verzweigung ein entsprechend gültiger Pfad ist, demnach also Modelle für die Eingabeformel impliziert. Auch wenn diese Art der Beweisführung besser verständlich sein könnte, bedingt sie, dass für deren Zwecke die Implementierung geändert werden muss.

Natürlich ist der hohe Aufwand der Beweisführung innerhalb von Isabelle/HOL nicht vergleichbar mit dem eines schriftlichen Beweises. Während die gesamte Beweisidee des Theorems 4.1 von Gerth et al. auf weniger als 3 DIN-A4-Seiten abgedruckt ist, umfasst der Isabelle-Beweis ca. 4500 Zeilen Code. Dies ist vor allem damit zu begründen, dass alle Eigenschaften, die in einem schriftlichen Beweis höchstens angedeutet werden, innerhalb von Isabelle/HOL akribisch hergeleitet werden müssen.

Kapitel 6

Codegenerierung

In diesem Kapitel geht es darum, aus der in Kapitel 5 vorgestellten Implementierung der LGBA-Konstruktion mit Hilfe von Isabelle/HOL ausführbaren Code zu generieren, der sich exakt so verhält, wie es die ursprüngliche Isabelle-Implementierung vorsieht. Dazu besitzt Isabelle einen Codegenerator [13], der in der Lage ist, bestimmte logische Spezifikationen in eine funktionale Programmiersprache wie OCaml [14] zu überführen. Im Wesentlichen handelt es sich dabei, um einen Prozess, der Datentyp,- Funktionsdefinition etc. an die jeweilige Sprache angepasst aus der Quellsprache, also der Spezifikationssprache von Isabelle, in die Zielsprache überführt.

6.1 Funktionsweise des Codegenerators

Die Funktionsweise des Codegenerators soll im Folgenden anhand von Beispielen verdeutlicht werden.

Beispiel In diesem Beispiel wird eine Baumstruktur definiert. Wie in Abschnitt 4.1.1 beschrieben wird mit Hilfe des Schlüsselworts `datatype` ein entsprechender Datentyp implementiert.

```
datatype
  'a tree = Leaf
          | Node 'a "'a tree" "'a tree"
```

Ein Knoten (`Node`) dieser Baumstruktur trägt eine Beschriftung sowie den linken und rechten Teilbaum des Knotens, während ein Blatt (`Leaf`) keine Beschriftung besitzt..

Nun soll eine Funktion definiert werden, die ausgehend von einer Beschriftung und zwei Bäumen einen neuen Baum zurück gibt.

```
definition concat :: "['a, 'a tree, 'a tree] ⇒ 'a tree"
where
  "concat l lt rt = Node l lt rt"
```

Die Funktion `concat` wird dieser Anforderung gerecht. Nun soll aus dieser Definition ausführbarer Code generiert werden. Für diese Zwecke existiert das Schlüsselwort **export_code**, das wie folgt verwendet wird.

```
export_code concat in OCaml file "concat.ml"
```

Mit dieser Anweisung wird die Implementierung der Funktion `concat` in OCaml-Code übersetzt und in die Datei `concat.ml` gespeichert. Der Inhalt dieser Datei sieht wie folgt aus:

```
module Beispiel =
struct

type 'a tree = Node of 'a * 'a tree * 'a tree | Leaf;;

let rec concat l lt rt = Node (l, lt, rt);;

end;; (*struct Beispiel*)
```

Bei der Ausgabe handelt es sich um ein OCaml-Modul, das in gewissen Sinne konsistent mit der entsprechenden Theoriedatei von Isabelle ist. Neben der Funktion `concat` selbst wird auch die entsprechende Datentypdefinition, die von `concat` verwendet, ausgegeben.

Nun soll eine Funktion definiert werden, die alle Beschriftungen eines Baumes zurück gibt.

```
fun labelset :: "'a tree ⇒ 'a set"
where
  "labelset Leaf = {}"
| "labelset (Node l lt rt)
   = {l} ∪ labelset lt ∪ labelset rt"
```

Die Funktion `labelset` verhält sich zwar bezüglich dieser Anforderung, hat aber im Kontext der Codegenerierung ein Problem: Beim Rückgabewert der Funktion handelt es sich um eine Menge. Da Mengen potentiell unendlich groß sein können, gibt es (noch) keine Möglichkeit (beliebige) Mengen direkt in der Zielsprache zu repräsentieren. Da jedoch die Menge aller Beschriftungen eines Baumes endlich ist, besteht die Option diese Menge als eine Liste darzustellen.

```
fun labels :: "'a tree ⇒ 'a list"
where
  "labels Leaf = []"
| "labels (Node l lt rt)
   = l # (labels lt @ labels rt)"
```

Die Funktion `labels` liefert in der Tat eine Liste zurück und kann nun mit Hilfe des Codegenerators in die Zielsprache überführt werden.

```
export_code label_list in OCaml file "labels.ml"
```

Die Ausgabedatei `labels.ml` hat daraufhin folgende Gestalt:

```
module List =
struct

let rec append
  x0 ys = match x0, ys with x :: xs, ys -> x :: append xs ys
    | [], ys -> ys;;

end;; (*struct List*)

module Beispiel =
struct

type 'a tree = Node of 'a * 'a tree * 'a tree | Leaf;;
```

```
let rec labels
  = function
    Node (l, lt, rt) -> l :: List.append (labels lt) (labels rt)
    | Leaf -> [];;

end;; (*struct Beispiel*)
```

Die rekursiv definierte Funktion `labels` resultiert in einer entsprechend geeigneten Form als OCaml-Funktion. Dabei wird das Pattern Matching beim Übergang in die Zielsprache entsprechend übernommen. Der syntaktische Zucker jedoch wird erwartungsgemäß aufgelöst. In diesem Beispiel erkennt man das an der Syntax für Listen: Die Infixnotation `@` wird entsprechend in die dazugehörige Definition umgeschrieben. Interessant ist außerdem die Tatsache, dass Listen beim Übergang auf Listen der Zielsprache abgebildet werden. Dazu ist der Codegenerator von Isabelle/HOL speziell vorkonfiguriert (vgl. [13, Abschnitt 1.5.6]).

Nun hat die Funktion `labels` aber einen kleinen Schönheitsfehler: Die Modellierung von Mengen durch Listen führt implizit die Eigenschaft ein, dass Listen mehrere gleiche Elemente enthalten können. Um diesem Problem entgegen zu wirken, ist eine Funktion hilfreich, die doppelte Elemente aus einer Liste entfernt:

```
fun remdups :: "'a list ⇒ 'a list"
where
  "remdups [] = []"
| "remdups (x#xs)
   = (if x ∈ set xs then remdups xs
      else x # remdups xs)"
```

Um zu überprüfen, ob ein Element `x` in der Restliste `xs` enthalten ist, verwendet diese Funktionsdefinition die Element-von-Beziehung auf Mengen. Da bekanntlich Mengen und damit Operationen auf Menge nicht direkt implementierbar sind, ergibt sich auf diese Weise ein Problem für die Codeerzeugung. Isabelle bietet jedoch die Möglichkeit während dem Codeerzeugungsprozess bestimmte Ausdrücke äquivalent umzuschreiben. Dazu formuliert man die entsprechende Äquivalenz in einem Lemma, beweist die Aussage und markiert dieses als für den Codeerzeugungsprozess relevant.

```
lemma [code inline]:
  "x ∈ set xs ⟷ x mem xs"
```

Das Markierung „`code inline`" legt gerade fest, dass während der Codegenerierung die linke Seite dieser Äquivalenz durch die rechte Seite ersetzt werden soll. Dabei ist die Infix-Funktion `mem`, die das Enthaltensein eines Elements in einer Liste testet, so definiert, dass daraus direkt ausführbarer Code erzeugt werden kann.

6.2 Übergang zu allgemeinen LTL-Formeln

6.2.1 Transformation in Negationsnormalform

Im Prinzip besteht bereits jetzt die Möglichkeit nach der Vorarbeit aus Kapitel 5 Code aus der Isabelle-Spezifikation zu generieren. Die Funktion `create_lgba` aus Abschnitt 5.3.2.2 erzeugt bereits einen LGBA aus einer LTL-Formel. Jedoch ist die Eingabeformel bisher in expliziter Negationsnormalform[1] (vgl. Abschnitt 5.2). Beliebige LTL-Formeln haben aber einen anderen Typ und können somit nicht als Argument für `create_lgba` verwendet werden. Um aus einer beliebigen LTL-Formel den entsprechenden LGBA konstruieren zu können, muss diese zunächst in die implizite Negationsnormalform[2] überführt werden. Für diesen Zweck wurde in Abschnitt 5.1.3 die Funktion `pushneg` definiert. Zusätzlich ist eine Funktion notwendig, die diese eine LTL-Formel in impliziter Negationsnormalform in eine explizite Form überführt.

```
fun transf_frml :: "LTL.frml ⇒ frml"
where
  "transf_frml true = true_n"
| "transf_frml false = false_n"
| "transf_frml prop(q) = prop_n(q)"
| "transf_frml (not prop(q)) = nprop_n(q)"
| "transf_frml (φ and ψ) = transf_frml φ and_n transf_frml ψ"
| "transf_frml (φ or ψ) = transf_frml φ or_n transf_frml ψ"
| "transf_frml (X φ) = X_n (transf_frml φ)"
| "transf_frml (φ U ψ) = transf_frml φ U_n transf_frml ψ"
| "transf_frml (φ V ψ) = transf_frml φ V_n transf_frml ψ"
```

[1] Eine LTL-Formel des Datentyps aus Abschnitt 5.3.2.2 ist in expliziter Negationsnormalform.

[2] Eine allgemeine LTL-Formel im Sinne von Abschnitt 5.1 heißt in impliziter Negationsnormalform, falls sie die Anforderungen gemäß Abschnitt 5.1.3 erfüllt.

Diese Funktionsdefinition[3] sieht in der Tat zunächst unvollständig aus. Wird als Eingabeparameter eine negierte Formel, die nicht gerade mit einer negierten Proposition identisch ist, übergeben, so kann die Funktion auf einem solchen Wert nicht weiter berechnet werden. Wenn man jedoch vorher die Eingabeformel mittels `pushneg` in die implizite Negationsnormalform überführt, besteht dieses Problem nicht mehr. Folglich kann man nun die Funktion, die ausgehend von einer beliebigen LTL-Formel den entsprechenden LGBA konstruiert, wie folgt definieren.

definition `create_lgba' :: "frml ⇒ node lgba"`
where
`"create_lgba' ψ ≡ create_lgba (transf_frml (pushneg ψ))"`

Obwohl man die Korrektheit dieser Konstruktion ausgehend von der Korrektheitseigenschaft für `create_lgba` in Theorem 4.1 direkt einsehen kann, bietet es sich an, diese Eigenschaft auf der Ebene von Isabelle/HOL herzuleiten.

6.2.2 Herleitung der Korrektheit

Zunächst einmal ist es erforderlich, zu zeigen, dass die Semantik beim Übergang von einer allgemeinen LTL-Formel in eine LTL-Formel in expliziter Negationsnormalform erhalten bleibt.

lemma `ltl_nnf_equiv:`
shows `"ξ ⊨ ψ ⟷ ξ ⊨ₙ transf_frml (pushneg ψ)"`

Da die Tatsache, dass die Funktion `pushneg` die Semantik erhält, wie in Abschnitt 5.1.3 beschrieben, bereits feststeht, bleibt nur noch zu zeigen, dass die Funktion `transf_frml` nichts an der Semantik ändert. Diese Aussage lässt sich im Wesentlichen induktiv zeigen, erfordert aber auch die Berücksichtigung des Lemmas `pushneg_neg_struct` aus Abschnitt 5.1.3, das gerade besagt, wie negierte Formeln nach Anwenden der Funktion `pushneg` aussehen können.

Für die Spezifikation der Korrektheitseigenschaft wurde in Theorem 4.1 (vgl. Abschnitt 5.3.3.2) die Funktion `get_props` verwendet, um aus der Eingabeformel, die als LTL-Formel in expliziter Negationsnormalform vorliegt, alle Propositionen zu extrahieren und um so implizit die Menge Prop festzulegen. Eine ähnliche Funktion ist nun für allgemeine LTL-Formeln erforderlich.

[3] Der Typ `LTL.frml` steht dabei für allgemeine LTL-Formeln aus Abschnitt 5.1.

```
fun get_props' :: "LTL.frml ⇒ prop set"
where
  "get_props' prop(p) = {p}"
| "get_props' (not φ) = get_props' φ"
| "get_props' (φ and ψ) = get_props' φ ∪ get_props' ψ"
| "get_props' (φ or ψ) = get_props' φ ∪ get_props' ψ"
| "get_props' (X φ) = get_props' φ"
| "get_props' (φ V ψ) = get_props' φ ∪ get_props' ψ"
| "get_props' (φ U ψ) = get_props' φ ∪ get_props' ψ"
| "get_props' _ = {}"
```

Entscheidend ist dabei, die Tatsache, dass die so definierte Funktion `get_props'` für eine beliebige LTL-Formel die gleiche Menge von Propositionen berechnet, wie die Funktion `get_props` für die entsprechende LTL-Formel in expliziter Negationsnormalform.

```
lemma props_nnf_eq:
  shows "get_props' ψ = set (get_props (transf_frml (pushneg ψ)))"
```

Mit den so definierten Funktionen und deren Eigenschaft lässt sich nun Theorem 4.1 für allgemeine LTL-Formeln wie folgt formulieren:

```
theorem T4_1':
  assumes "∀i. ξ i ∈ Pow (get_props' φ)"
    shows "lgba_accept (create_lgba' φ) ξ ⟷ ξ ⊨ φ"
```

Dieses modifizierte Theorem folgt direkt aus Theorem 4.1 (vgl. Abschnitt 5.3.3.2) und den oben hergeleiteten Eigenschaften.

6.3 Codeerzeugung für LGBA-Konstruktion

Ausgehend von der Tatsache, dass sich die LGBA-Konstruktion, gegeben durch die Funktion `create_lgba'`, gemäß ihrer Spezifikation verhält, lässt sich nun unter der Annahme, dass der Codegenerator von Isabelle fehlerfrei funktioniert, Code aus dieser Konstruktion ableiten, der sich ebenfalls gemäß seiner Spezifikation korrekt verhält. Zuvor ist es jedoch erforderlich, Operationen auf endlichen Mengen, die in der Definition von `expand` aus Abschnitt 5.2.3.4 verwendet werden,

äquivalent so umzuschreiben, dass daraus ausführbarer Code generiert werden kann.

Wie die Element-von-Beziehung für Mengen entsprechend äquivalent umgeschrieben wird, wurde am Anfang von Kapitel 6 bereits anhand eines Beispiels erläutert. Eine weitere Operation auf endlichen Mengen, die in der Definition von `expand` verwendet wird, ist der Gleichheitstest. Dieser kann auf die Teilmengenbeziehung reduziert werden, die wie folgt implementiert wird:

```
lemma [code inline]:
  shows "set xs ⊆ set ys ⟷ list_all (λx. x mem ys) xs"
```

Dabei ist die Funktion `list_all` für die Argumente `P` und `xs` so definiert, dass der Rückgabewert dieser Funktion nur dann `True` ergibt, wenn das Prädikat `P` für alle Elemente der Liste `xs` erfüllt ist.

Die Teilmengenbeziehung kann nun zur Implementierung des Gleichheitstests auf Mengen herangezogen werden:

```
lemma [code inline]:
  shows "set xs = set ys ⟷ set xs ⊆ set ys ∧ set ys ⊆ set xs"
```

Unter Verwendung dieser Identität liefert das folgende Kommando den entsprechenden OCaml-Code für die Funktion `create_lgba'`:

```
export_code create_lgba' in OCaml file "ltl2lgba.ml"
```

Der Code der Datei `ltl2lgba.ml` kann nun mit Hilfe des OCaml-Compilers ausführbar gemacht werden.

6.4 Demo-Anwendung

Zu Demonstrationszwecken habe ich basierend auf der OCaml-Quelldatei, die als Ausgabe des Codegenerierungsprozesses aus Abschnitt 6.3 hervorgeht, eine fertige Anwendung konzipiert. Diese erwartet als Eingabe eine LTL-Formel (als String) und liefert den entsprechenden LGBA (in textueller Form) aus. Im Wesentlichen basiert das Konzept dieser Anwendung auf drei Modulen, wie folgende Abbildung zeigt:

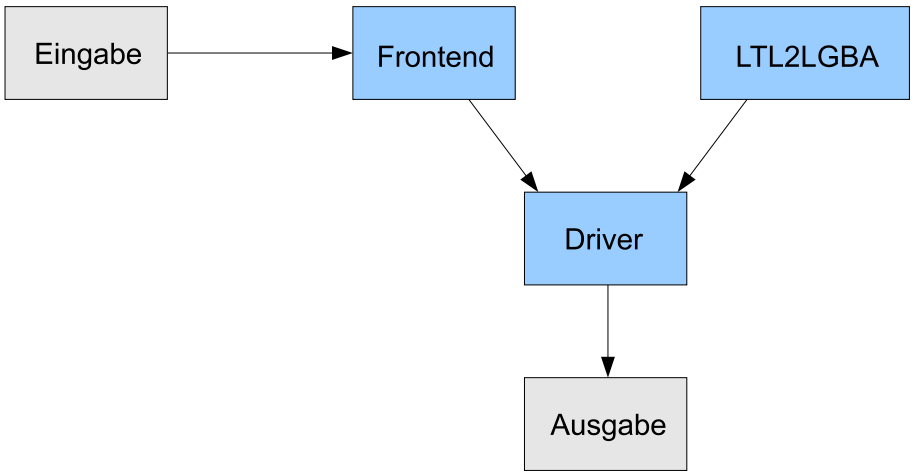

Abbildung 6.1: Konzept der Anwendung

Während das Modul **LTL2LGBA** vollkommen automatisch durch den Codegenerator von Isabelle erzeugt wird, besteht das **Frontend** sowie der **Driver** aus Code, der außerhalb von Isabelle erstellt wurde. Dabei setzt sich das **Frontend** aus einem Lexer und einem Parser zusammen. Beim Lexer (oft auch Scanner genannt) handelt es sich hauptsächlich um eine Funktion, die einen Eingabestring in Sinneinheiten unterteilt und auf so genannte Terminalsymbole abbildet. Diese Terminalsymbole sind Repräsentanten der jeweiligen Sinneinheiten. Der Parser wiederum analysiert die Terminalsymbole, die der Lexer liefert, bzgl. einer Grammatik, ob es sich um eine gültige Eingabe handelt oder nicht. Sowohl den Lexer als auch den Parser habe ich mit Hilfe des Lexer- bzw. Parsergenerators für OCaml (ocamllex, ocamlyacc) realisiert. Solche Generatoren erlauben es, ausgehend von einer einfachen Spezifikationssprache einen Lexer bzw. einen Parser automatisch erstellen zu lassen.

Die Spezifikation des Lexers besteht hauptsächlich aus einer regulären Grammatik:

```
let ident = ['a'-'z' ] ['a'-'z' 'A'-'Z' '_' '0'-'9']*

rule token = parse
    [' ' '\t' '\r']+       { token lexbuf }
  | '~'                    { NOT }
  | 'X'                    { NEXT }
```

```
  | 'F'                  { FINAL }
  | 'G'                  { GLOBAL }
  | '|' '|'              { OR }
  | '&' '&'              { AND }
  | '-' '>'              { IMPLIES }
  | 'U'                  { UNTIL }
  | '('                  { LPARENT }
  | ')'                  { RPARENT }
  | ident as id          { scan_ident id }
  | eof                  { EOF }
```

Die Symbole `NOT`, `NEXT` usw. sind gerade Terminalsymbole. Beim Verarbeiten des Eingabestrings wird etwa das Zeichen `X` auf das Terminalsymbol `NEXT` abgebildet. Man erhält folglich für eine gültige Eingabe eine Folge von Terminalsymbolen. Die Hilfsdefinition `ident` legt gerade fest, wann eine Zeichenkette als Bezeichner betrachtet wird. Die Hilfsfunktion `scan_ident` erkennt dabei die zwei ausgezeichneten Bezeichner `true` und `false`:

```
let scan_ident id =
  match id with
    "true" -> TRUE
  | "false" -> FALSE
  | _ -> IDENT (str_to_cl id)
```

Die Funktion `str_to_cl` überführt einen String in eine entsprechende Liste von Buchstaben, da Isabelle Strings als solche Listen repräsentiert. Auf diese Weise wird bereits jetzt der Übergang zu LTL-Formeln des Moduls **LTL2LGBA** vorbereitet.

Nachdem der Lexer den Eingabestring in die entsprechende Folge von Terminalsymbolen erfolgreich überführt hat, übergibt dieser die Ausgabe an den Parser. Der Parser ordnet gemäß einer kontextfreien Grammatik den Terminalsymbolen eine LTL-Formel im Sinne der Isabelle-Definition zu.

```
input: formula EOF { $1 }

formula: IDENT                  { LTLProp $1 }
       | TRUE                   { LTLTrue }
       | FALSE                  { LTLFalse }
```

```
| NOT formula             { LTLNeg $2 }
| NEXT formula            { LTLNext $2 }
| FINAL formula           { LTLUntil (LTLTrue, $2) }
| GLOBAL formula          { LTLNeg
                              (LTLUntil
                                 (LTLTrue, LTLNeg $2)) }
| formula OR formula      { LTLOr ($1, $3) }
| formula AND formula     { LTLAnd ($1, $3) }
| formula IMPLIES formula { LTLOr (LTLNeg $1, $3) }
| formula UNTIL formula   { LTLUntil ($1, $3) }
| LPARENT formula RPARENT { $2 }
```

Die Nichtterminale dieser Grammatik sind `input` und `formula`. Gleichzeitig ist `input` das Startsymbol. Die Terminalsymbole sind die Terminale der Grammatik. Jede rechte Regelseite wird mit einer LTL-Formel gleichgesetzt. Dabei referenziert `$n` das `n`-te Grammatiksymbol der rechten Regelseite.

Mit dem **Frontend** besteht nun die Möglichkeit einen String in eine LTL-Formel abzubilden. Die LTL-Formel kann dann direkt an die Funktion `create_lgba` des Moduls **LTL2LGBA** übergeben werden. Diese Arbeit und zusätzlich die textuelle Ausgabe des resultierenden LGBA erledigt das Modul **Driver**.

Beispiel Das kompilierte Programm aufgerufen mit der Eingabeformel „`p U q`" (vgl. Abbildung 5.2 auf Seite 57) liefert folgende Ausgabe:

```
INCOMING: [2, 1]
TRANSITIONS: [3->3, 2->3, 1->2, 1->1]
ACCEPTANCE SETS: [[3, 2]]
LABELS: {3->[[], [q], [p], [p, q]], 2->[[q], [p, q]], 1->[[p], [p, q]]}
```

Analog zur Abbildung 5.2 muss der Pfad, der stets im Zustand `1` verbleibt, zurückgewiesen werden. Die Akzeptanzmenge sorgt dafür, dass dies geschieht. Die Beschriftung des Zustands `1` fordert, dass zumindest die Proposition `p` erfüllt sein muss. Gleiches gilt für die Beschriftung des Zustands `2` und die Proposition `q`. In Zustand `3` dagegen wird keinerlei Forderung an die Erfüllbarkeit einer Proposition gestellt. Aus diesem Grund sind auch nur Zustände `1` und `2` Startzustände. Während mit Zustand `2` Pfade beginnen, in denen bereits zu Beginn `q` erfüllt ist, erfordern Pfade, die mit Zustand `1` beginnen, zunächst die Erfüllbarkeit von `p`.

Tests Ich habe die Demo-Anwendung auf so genannten Formeln mit n Fairnessbedingungen der Form $\neg((\mathsf{G}\,\mathsf{F}\,\mathsf{p}_1 \wedge ... \wedge \mathsf{G}\,\mathsf{F}\,\mathsf{p}_\mathsf{n}) \rightarrow \mathsf{G}\,(\mathsf{q} \rightarrow \mathsf{F}\,\mathsf{r}))$ für $\mathsf{n} = 1$ bis $\mathsf{n} = 6$ getestet. Während die Ausführung für $\mathsf{n} \leq 5$ auf meinem Notebook mit einer 1.7 GHz CPU und 1 GB RAM stets unter einer Minute blieb, benötigte die Berechnung für $\mathsf{n} = 6$ ca. 12 min. In Anbetracht der Ergebnisse, die Gastin und Oddoux [16] in ihrer Arbeit festgestellt haben, ist das ein relativ guter Wert.

Kapitel 7

Schlusswort

In dieser Diplomarbeit habe ich das Verfahren von Gerth et al. [1] zur Erzeugung von Büchi-Automaten aus LTL-Formeln mit Hilfe des interaktiven Theorembeweisers Isabelle/HOL implementiert und nachgewiesen, dass meine Implementierung entsprechende Korrektheitseigenschaften erfüllt. Des weiteren habe ich gezeigt, dass aus meiner Implementierung mit Hilfe des Codegenerators von Isabelle Code erzeugt werden kann, um daraus ausführbare Programme konstruieren zu können. Im Wesentlichen handelt es sich bei dieser Vorgehensweise um einen brauchbaren Ansatz, um zuverlässige Programme entwickeln zu können. Jedoch gilt es dafür eine Reihe schwer überwindbarer Hürden zu nehmen. Eine solche Implementierung erfordert nämlich eine hohe Zeitinvestition. Gegenüber üblichen Herangehensweisen Programme zu entwickelt, gilt es eine Reihe von Korrektheitseigenschaften wie etwa die Terminierung von (rekursiven) Funktionen nachzuweisen. Je nach Komplexität der Implementierung kann das beliebig viel Zeit in Anspruch nehmen. Insgesamt habe ich für meine Implementierung gute 3 Monate benötigt. Grob geschätzt umfassen die dazugehörigen Theoriedateien 6800 Zeilen Code. Und es besteht noch Optimierungspotential: Die Repräsentation der Mengen könnte man etwa durch sortierte Listen vornehmen, um die Effizienz der Implementierung zu erhöhen. Weiterhin sollte das Design der Implementierung nach Möglichkeit von Anfang an feststehen. Besteht Bedarf dieses im Nachhinein, also nachdem sämtliche Korrektheitsbeweise geführt worden sind, zu ändern, so sieht man sich in der Regel vor die Aufgabe gestellt, diese Beweise entsprechend modifizieren zu müssen. Bereits eine geringe Anpassung einer umfangreicheren Implementierung kann einen hohen Änderungsaufwand der Beweisführung nach sich ziehen.

In jedem Fall lohnt sich diese Vorgehensweise der Softwareentwicklung für Funk-

tionen, die besonders kompliziert sind und deren Korrektheit mit bloßem Hinsehen nur schwer einsehbar ist. Auch in Einsatzgebieten wie dem Modellprüfen, in denen eine zuverlässige Funktionsweise der Software zwingend erforderlich ist, kann diese Vorgehensweise von Vorteil sein. Ich könnte mir dabei vorstellen, dass man klassische Entwicklungsumgebungen mit Isabelle/HOL kombiniert. Schwer einsehbare Programmteile würde man dann mit Isabelle/HOL implementieren, verifizieren und in Code überführen. Den resultierenden Code könnte man schließlich als eine Art Bibliothek betrachten, um diese im klassischen Sinne weiter zu verarbeiten.

Literaturverzeichnis

[1] Rob Gerth, Doron Peled, Moshe Y. Vardi, Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, Proceedings of the 15th International Symposium on Protocol Specification, Testing, and Verification, volume 38 of IFIP Conference Proceedings, pages 3-18. Chapman & Hall, 1996.

[2] Edmund M. Clarke, Orna Grumberg, Doron A. Peled. Model checking. MIT Press, 2007.

[3] Pierre Wolper, Moshe Y. Vardi, Aravinda P. Sistla. Reasoning about infinite computation paths. Proc. 24th IEEE Symp. on Foundations of Computer Scence. Tuscon, 1983.

[4] Yonit Kesten, Zohar Manna, Hugh McGuire, Amir Pnueli. A decision algorithm for full propositional temporal logic. CAV, pages 97-109. Springer-Verlag, 1993.

[5] Christel Baier, Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008.

[6] Moshe Y. Vardi, Pierre Wolper. Reasoning about infinite computations. Information and Computation, 1994.

[7] Edmund M. Clarke, Ernest A. Emerson, Aravinda P. Sistla: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM, 1986.

[8] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design Vol. 1, 1992.

[9] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. Isabelle/HOL: A proof assistant for higher-order logic. Springer-Verlag, 2002.

[10] Joachim Lambek, Philip J. Scott. Introduction to higher-order categorical logic. Cambridge University Press, 1994.

[11] Makarius Wenzel. The Isabelle/Isar reference manual. 2008. `http://isabelle.in.tum.de/dist/Isabelle/doc/isar-ref.pdf`

[12] Alexander Krauss. Defining recursive functions in Isabelle/HOL. `http://isabelle.in.tum.de/dist/Isabelle/doc/functions.pdf`

[13] Florian Haftmann. Code generation from Isabelle/HOL theories. 2008. `http://isabelle.in.tum.de/dist/Isabelle/doc/codegen.pdf`

[14] Joshua B. Smith. Practical OCaml. Apress, 2006.

[15] Jean Louis Krivine, René Cori. Lambda-calculus: types and models. Ellis Horwood, 1993.

[16] Paul Gastin, Denis Oddoux. Fast LTL to Büchi Automata Translation. Springer-Verlag, 2001.