UNIVERSITÄT ZU LÜBECK
INSTITUTE OF SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Transformation von regulärer Linearzeit-Temporallogik zu Paritätsautomaten

Malte Schmitz,
Lübeck im Januar 2012

korrigierte Fassung,
Lübeck im März 2014

Diese Bachelorarbeit wurde ausgegeben und betreut von
Prof. Dr. Martin Leucker

Institut für Softwaretechnik und Programmiersprachen
Universität zu Lübeck

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

(Malte Schmitz)
Lübeck, 20. Januar 2012

Kurzfassung Reguläre Linearzeit-Temporallogik (RLTL) vereinigt die Vorteile von linearer Temporallogik (LTL) und ω-regulären Ausdrücken in einer neuen Logik. Um RLTL in der Praxis verwenden zu können, wird eine effiziente Umwandlung von RLTL zu Automaten benötigt. Im Rahmen dieser Arbeit wurde ein Scala-Programm implementiert, das RLTL-Formeln einliest und zu endlichen alternierenden Zwei-Wege-Paritätsautomaten umwandelt. In dieser Arbeit wird die verwendete Umwandlung und deren Implementierung detailliert beschrieben.

Abstract Regular linear temporal logic (RLTL) combines the advantages of linear temporal logic (LTL) and ω-regular expressions in a new logic. To use RLTL in practise an efficient translation from RLTL into automata is needed. In the context of this thesis such a translation was implemented as a scala program, which reads in RLTL formulas and translates them into finite alternating two-way parity automata. In this thesis the used translation and its implementation will be described in detail.

Inhaltsverzeichnis

1. Einleitung

Eines der zentralen Probleme in der praktischen Softwareentwicklung ist die Frage, ob ein entwickeltes Softwaresystem alle Spezifikationen erfüllt und fehlerfrei arbeitet. Während in vielen Fällen willkürliches manuelles Ausprobieren von Funktionalitäten oder systematischeres und kleinteiligeres Ausprobieren durch Testfälle ausreicht, so existieren auch eine Vielzahl verschiedener Computersysteme, bei denen für den Nachweis der Fehlerfreiheit mehr Arbeit investiert werden muss. Je größer der mögliche Schaden durch eine Fehlfunktion ist, umso wichtiger wird es, Fehler in einem Computersystem zu finden.

Eine Technik zum Nachweis der korrekten Funktionalität eines Systems ist das Verfahren der Modellprüfung (engl. Model Checking). Dabei wird vollautomatisch überprüft, ob eine formale Systembeschreibung (ein formales Modell des Systems) eine logische Eigenschaft erfüllt. Die Eingabe für eine Modellprüfung besteht aus einem Modell und einer Spezifikation. Erfüllt das Modell die Spezifikation, so kann der Modellprüfer dies bestätigen. Im anderen Fall zeigt der Modellprüfer auf, an welcher Stelle das Modell die Spezifikation verletzt. Die Spezifikation wird dabei häufig in Form von temporal-logischen Formeln angegeben. Für ein Modell M und eine temporal-logische Formel φ kann dann eine automatenbasierte Modellprüfung durchgeführt werden. Dazu werden zunächst das Modell M und die Formel φ in zwei Büchi-Automaten $B(M)$ und $B(\varphi)$ umgewandelt. Nun gilt $M \models \varphi$ (das Modell M erfüllt die Formel φ) genau dann, wenn für die von den Büchi-Automaten akzeptierten Sprachen $L(B(M))$ bzw. $L(B(\varphi))$

$$L(B(M)) \subseteq L(B(\varphi))$$

gilt, wenn also die von $B(M)$ akzeptierten Worte nur eine Teilmenge der von $B(\varphi)$ erlaubten Worte sind. Für zwei Mengen X und Y gilt die Beziehung $X \subseteq Y$ genau dann, wenn $X \cap \overline{Y} = \emptyset$ gilt. Statt $\overline{L(B(\varphi))}$ kann auch der gleichwertige Ausdruck $L(B(\neg\varphi))$ betrachtet werden. Die zu überprüfende Aussage ist damit gerade äquivalent zu

$$L(B(M)) \cap L(B(\neg\varphi)) = L(B(M) \cap B(\neg\varphi)) = \emptyset$$

Es bleibt also zu überprüfen, ob der Produktautomat zu den Automaten $B(M)$ und $B(\neg\varphi)$ nur die leere Sprache akzeptiert. Ein solcher Leerheitstest kann auf Büchi-Automaten sehr einfach durchgeführt werden: Sind alle erreichbaren Zyklen von

Transitionen frei von akzeptierenden Zuständen, so akzeptiert der Automat kein Wort.

Ein ganz anderer Ansatz wird bei der Laufzeitverifikation (engl. Runtime Verfication) verfolgt: Hier geht es darum, die korrekte Funktion eines Systems zu seiner Laufzeit nachzuweisen. Diese Technik ist insbesondere dann nützlich, wenn der Quellcode bzw. ein anderes Modell der zu überprüfenden Software nicht zur Verfügung stehen. Stattdessen werden Seiteneffekte der laufenden Software, wie zum Beispiel Reservierung und Freigabe von Speicher, betrachtet. So sehr sich dieser Ansatz auch von der Modellprüfung unterscheidet, wird doch auch hier eine formale Spezifikation der zu prüfenden Eigenschaften als Formel einer temporalen Logik benötigt. Ganz ähnlich zur Modellprüfung wird auch bei der Laufzeitverifikation die Spezifikation in Formelschreibweise in einen Büchi-Automaten übersetzt, der zur vollautomatischen Überprüfung der Spezifikation verwendet wird.

Beide Verfahren sind Beispiele praktischer Anwendungen von temporal-logischen Formeln in der Softwareverifikation. Als Temporallogik wird häufig die lineare Temporallogik (LTL) verwendet, da mit LTL eine relativ intuitive Spezifikation zeitlicher Zusammenhänge über linearen Pfaden möglich ist. LTL ist allerdings eine echte Teilmenge der ω-regulären Ausdrücke, da LTL unter anderem keine Wiederholungen spezifizieren kann. Trotz ihrer größeren Mächtigkeit werden ω-reguläre Ausdrücke in der Praxis nicht gerne eingesetzt, da mit ihnen keine Konjunktion und keine Negation möglich sind. Daher kann die Spezifikation zeitlicher Zusammenhänge über linearen Pfaden mit ω-regulären Ausdrücken schnell sehr unübersichtlich und wenig intuitiv werden. Als Lösung für dieses Problem wird in [LS07] zum ersten Mal die reguläre lineare Temporallogik (RLTL) vorgestellt. Diese Logik hat die Mächtigkeit von ω-regulären Ausdrücken und sowohl ω-reguläre Ausdrücke als auch LTL-Formeln können in RLTL-Formeln gleicher Größe umgewandelt werden. RLTL vereint in diesem Sinne die Vorteile von LTL und ω-regulären Ausdrücken in einer neuen Logik.

Wie aus obigen Anwendungsbeispielen ersichtlich wird, benötigt man eine effiziente Umwandlung von RLTL zu Büchi-Automaten, um RLTL in der Praxis verwenden zu können. Eine solche Umwandlung wird in [SSF11] beschrieben. Dabei wird der RLTL-Ausdruck zunächst in einen alternierenden Paritätsautomaten umgewandelt und dieser dann in einen Büchi-Automaten. In dieser Arbeit wurde diese Umwandlung von RLTL zu alternierenden Paritätsautomaten implementiert. Die weiterführende Umwandlung von alternierenden Paritätsautomaten zu Büchi-Automaten wurde parallel und in enger Zusammenarbeit mit dieser Arbeit in [Sch11] realisiert. Zusammen ergibt sich eine Programmkette, die RLTL-Formeln in nichtdeterministische Büchi-Automaten umwandeln kann. Diese können dann in Modellprüfer-Paketen verwendet werden.

Da in dieser Arbeit detaillierte Definitionen der verwendeten Automaten und Logiken benötigt werden, werden zu Beginn in Kapitel 2 die Formeln und in Kapitel 3 die

Automaten eingeführt und definiert, bevor in Kapitel 4 die eigentliche Umwandlung detailliert erläutert wird. Auch wenn in den ersten Kapiteln allgemeingültige Definitionen gegeben werden, wird hier die folgende Umwandlung in vielen Details bereits vorbereitet. Insbesondere werden bereits Grammatiken für die logischen Formeln eingeführt, die für den Eingabeparser nahezu direkt verwendet werden. Die Darstellung der für die Umwandlung nötigen Schritte in Kapitel 4 beginnt mit der Umwandlung von regulären Ausdrücken zu nichtdeterministischen endlichen Automaten. Auf diesen Automaten wird dann die Entfernung von ε-Transitionen ausführlich untersucht, da diese für die weitere Verarbeitung benötigt wird. Weiter wird die Umwandlung von RLTL zu alternierenden Paritätsautomaten beschrieben und anschließend werden verschiedene Möglichkeiten betrachtet, den entstandenen Automaten weiter zu optimieren. In Kapitel 5 wird schließlich die Realisierung der Umwandlung als Scala-Programm dargestellt. Dort werden die verwendeten Datenstrukturen, die Ein- und Ausgabeformate, die Parserstrukturen und die Verwendung des entstandenen Kommandozeilenwerkzeugs beschrieben und die praktischen Tests mit diesem Wekrzeug erläutert. Am Ende wird in Kapitel 6 eine Zusammenfassung gegeben und es werden die weiteren Schritte analysiert, die zum praktischen Einsatz der im Rahmen dieser Arbeit entwickelten Software nötig sind.

2. Formeln

In diesem Kapitel wird neben anderen Formeln bzw. Logiken mit regulärer Linearzeit-Temporallogik (RLTL) die wichtigste Logik dieser Arbeit eingeführt. Wir werden sehen, dass RLTL die gleiche Mächtigkeit wie ω-reguläre Ausdrücke besitzt, durch die einfache Umwandlung von der weniger mächtigen Linearzeit-Temporallogik (LTL) zu RLTL und durch eigene Operatoren für Negation und Konjunktion aber viele praktische Vorteile gegenüber ω-regulären Ausdrücken besitzt.

2.1. Reguläre Ausdrücke

Eine reguläre Sprache ist eine Menge endlicher Wörter über einem Eingabealphabet Σ. Eine solche *endliche Sprache* ist eine Sprache des Typs 3 in der Chomsky-Hierarchie und damit die unterste Stufe. Zur Beschreibung regulärer Sprachen werden reguläre Ausdrücke verwendet. Reguläre Ausdrücke haben die gleiche Mächtigkeit wie endliche Automaten in Bezug auf die mögliche Komplexität der Menge der akzeptierten Worte. Jeder reguläre Ausdruck kann somit in einen endlichen Automaten umgewandelt werden und jeder endliche Automat in einen regulären Ausdruck.

Genau wie ω-reguläre Ausdrücke basiert auch RLTL auf regulären Ausdrücken. Die durch den regulären Ausdruck definierten endlichen Segmente werden jeweils durch Pump-Operatoren in die Unendlichkeit gehoben, sodass Sprachen auf unendlichen linearen Pfaden definiert werden können.

2.1.1. Syntax

Die folgende Grammatik in Erweiterter Backus-Naur-Form (EBNF) definiert die Syntax regulärer Ausdrücke für endliche Worte:

```
RE = T { "|" T }      // Disjunktion
 T = K { K }          // Konkatenation
 K = E "*" E | E      // Kleene-Operator
 E = P | "(" RE ")"   // Klammerung
```

Dabei ist `P` ein Basisausdruck, der aus einer booleschen Kombination von Propositionen aus einer endlichen Grundmenge besteht. Ein solcher Ausdruck wird in einem einzelnen Zustand bzw. dem Schritt zwischen zwei Zuständen ausgewertet. Ein Basisausdruck kann auch als Element des Eingabealphabets verstanden werden, wobei true als Symbol für alle Propositionen und false als Symbol für die leere Menge im Alphabet enthalten sind. Somit steht true für eine beliebige Eingabe und liest jedes Zeichen, während false zu keinem Zeichen der Eingabe passt.

Der Operator | ist als klassische Disjunktion zu verstehen. Die Konkatenation hingegen kommt, wie in vielen praktischen Anwendungen regulärer Ausdrücke üblich, ohne einen expliziten Operator aus und wird einfach durch das Hintereinanderschreiben von regulären Ausdrücken realisiert. Der Operator $*$ ist der binäre Kleene-Stern. Der Ausdruck vor dem Stern muss dabei beliebig oft (also auch kein mal) akzeptieren, während der Ausdruck nach dem Stern genau einmal akzeptieren muss. Der Ausdruck nach dem Stern ist verpflichtend und kann nicht weggelassen werden.

Diese Grammatik ist in dem Sinne eindeutig, dass es nur eine Möglichkeit gibt, ein Wort aus der Sprache der regulären Ausdrücke abzuleiten. So ist die Rangfolge der Operatoren durch die Grammatik festgelegt und die Grammatik kann direkt zur Implementierung des Parsers verwendet werden (siehe Abschnitt 5.4 auf Seite 85).

2.1.2. Semantik

Die hier verwendete Version regulärer Ausdrücke beschreibt endliche Segmente von unendlichen Wörtern. Auf diese Weise kann die Semantik besser in die folgenden Semantiken eingebaut werden.

Ein unendliches Wort $w \in \Sigma^\omega$ ist eine unendliche Folge von Elementen des Eingabealphabets Σ (im Folgenden Zeichen genannt) und kann auch als unendlicher linearer Pfad verstanden werden. Eine Position $i \in \mathbb{N}$ in einem Wort ist eine natürliche Zahl, wobei das erste Zeichen des Wortes an Position 0 steht. $w[i]$ bezeichnet das Zeichen des Wortes w an der Position i. Ein Segment eines Wortes ist ein Tupel (w, i, j) aus einem Wort w und zwei Positionen i und j, die das erste und das letzte Zeichen des Segments beschreiben. Ein punktiertes Wort ist ein Tupel (w, i) bestehend aus einem Wort und einer Position i des ersten Zeichens. Dabei ist zu beachten, dass ein Segment und ein punktiertes Wort nicht aus abgeschnittenen Worten, sondern aus den Positionen und dem ganzen Wort bestehen.

Die Semantik ist formal definiert als Relation $\models_{\mathrm{RE}}$ zwischen Segmenten und regulären Ausdrücken. Diese Relation wird induktiv in Tabelle 2.1 auf der nächsten Seite definiert, wobei a und b reguläre Ausdrücke, p ein Zeichen des Alphabets, i und j Positionen und w ein unendliches Wort seien.

Relation	Semantik
$(w, i, j) \models_{\text{RE}} p$	$w[i]$ erfüllt p und es gilt $j = i + 1$.
$(w, i, j) \models_{\text{RE}} a \mid b$	Entweder gilt $(w, i, j) \models_{\text{RE}} a$ oder $(w, i, j) \models_{\text{RE}} b$ oder beides.
$(w, i, j) \models_{\text{RE}} ab$	Es existiert ein k, sodass $(w, i, k) \models_{\text{RE}} a$ und $(w, k, j) \models_{\text{RE}} b$ gelten.
$(w, i, j) \models_{\text{RE}} a * b$	Entweder gilt $(w, i, j) \models_{\text{RE}} b$ oder es existiert eine Folge $(i_0, i_1, \ldots, i_m)$ mit $i_0 = i$, sodass für alle $k < m$ die Beziehung $(w, i_k, i_{k+1}) \models_{\text{RE}} a$ gilt und schließlich $(w, i_m, j) \models_{\text{RE}} b$ gilt.

Tabelle 2.1.: Induktive Definition der Relation $\models_{\text{RE}}$ zur Beschreibung der Semantik regulärer Ausdrücke. Die Relation gilt genau dann, wenn die angegebene Semantik gilt.

Diese alternative Syntax kann bis auf eine Ausnahme mit den klassischen regulären Ausdrücken in Einklang gebracht werden. Da der der Kleene-Operator hier als binärer Operator definiert ist, kann das leere Wort nicht in auf diese Weise definierten Sprachen enthalten sein. Ein regulärer Ausdruck x definiert eine Sprache $\mathcal{L}(x) \subseteq \Sigma^+$ indem $v \in \mathcal{L}(x)$ genau dann gilt, wenn für ein beliebiges unendliches Wort $w \in \Sigma^\omega$ die Relation $(vw, 0, |v|) \models_{\text{RE}} x$ gilt.

2.1.3. Reguläre Ausdrücke mit Vergangenheit

Um im weiteren Verlauf dieser Arbeit RLTL mit Vergangenheit basierend auf regulären Ausdrücken definieren zu können, führen wir hier einen weiteren Operator ein: Den Vergangenheits-Operator auf Basisausdrücken. Die folgende Definition benötigt zum ersten Mal, dass die Relation $\models_{\text{RE}}$ auf Sequenzen, also Tupeln (w, i, j) aus einem unendlichen Wort w und zwei Positionen i und j, definiert ist und nicht auf endlichen Wortabschnitten. Insbesondere ist dabei auch $i \geq j$ erlaubt.

Es gilt $(w, i, j) \models_{\text{RE}} -p$, genau dann wenn $w[i]$ den Basisausdruck p erfüllt und $j = i - 1$.

Damit lässt sich zum Beispiel der Ausdruck

$$\text{notfirst} := -\,\text{true}\,\text{true}$$

definieren. Alle Segmente (w, i, i) mit $i \neq 0$ erfüllen diesen Ausdruck.

Dieser Vergangenheits-Operator auf Basisausdrücken kann verwendet werden, um einen beliebigen regulären Ausdruck umzuwandeln in einen in die Vergangenheit gerichteten Ausdruck. Dazu führen wir einen erweiterten Vergangenheits-Operator auf regulären Ausdrücken ein und zeigen, wie dieser auf den Vergangenheits-Operator auf Basisausdrücken zurückgeführt werden kann.

Der Operator $\cdot^{-1}$ wird induktiv über folgende Äquivalenzen definiert. Als Induktionsanfang gilt für einen Basisausdruck p

$$\begin{aligned} p^{-1} &:\equiv -p \\ (-p)^{-1} &:\equiv p \end{aligned}$$

und als Induktionsschritt gilt für die regulären Ausdrücke a und b

$$\begin{aligned} (a \mid b)^{-1} &:\equiv a^{-1} \mid b^{-1} \\ (ab)^{-1} &:\equiv b^{-1}a^{-1} \\ (a * b)^{-1} &:\equiv b^{-1} \mid b^{-1}a^{-1} * a^{-1} \end{aligned}$$

Bei der folgenden Definition von RLTL wird kein eigener Vergangenheits-Operator definiert. Vielmehr wird RLTL mit Vergangenheit definiert, indem reguläre Ausdrücke mit Vergangenheit als Operanden in RLTL-Formeln zugelassen werden.

2.1.4. Vollständige Syntax

Mit den beiden im letzten Abschnitt neu eingeführten Operatoren ergibt sich folgende Grammatik der Syntax regulärer Ausdrücke für endliche Worte:

```
RE = T { "|" T }          // Disjunktion
 T = K { K }              // Konkatenation
 K = E "*" E | E          // Kleene-Operator
 E = F "⁻¹" | "−" P | F   // Vergangenheit
 F = P | "(" RE ")"       // Klammerung
```

Dabei ist `P` wieder ein Basisausdruck, wie oben beschrieben.

2.2. ω-reguläre Ausdrücke

ω-reguläre Sprachen sind eine Menge unendlicher Worte über einem Eingabealphabet Σ. So wie alle regulären Sprachen von einem endlichen Automaten auf endlichen Worten akzeptiert werden können, können alle ω-regulären Sprachen von einem Büchi-Automaten auf unendlichen Worten akzeptiert werden. Die Menge aller ω-regulärer Sprachen ist daher gerade die Menge der büchi-erkennbaren Sprachen. ω-reguläre Ausdrücke sind entsprechend analog zu regulären Ausdrücken in der Lage alle ω-regulären Sprachen zu definieren. In dieser Arbeit werden ω-reguläre Ausdrücke nur als theoretischer und praktischer Nachweis der Mächtigkeit von RLTL benötigt.

2.2.1. Syntax

Die folgende Grammatik in EBNF definiert die Syntax ω-regulärer Ausdrücke für unendliche Worte:

```
ORE = O { "||" O }    // Disjunktion
  O = RE ";" RE "ω"   // Pump-Operator in die Unendlichkeit
```

Dabei steht RE für die Grammatik eines oben definierten regulären Ausdrucks auf endlichen Worten. Der Operator $\|$ realisiert die klassische Disjunktion. Die Unterscheidung zwischen den Operatoren $\|$ und $|$ ist wichtig, da ersterer auf ω-regulären Ausdrücken operiert und letzterer auf regulären Ausdrücken. Obwohl ω-reguläre Ausdrücke die regulären Ausdrücke erweitern, wird ein eigener Disjunktions-Operator benötigt. Durch diesen neuen Operator werden Ausdrücke verodert, die unendliche Worte akzeptieren, während der bereits vorhandene Disjunktions-Operator der regulären Ausdrücke über Ausdrücken definiert ist, die endliche Worte akzeptieren. Der zweite neue Operator besteht aus zwei Operanden und ist sehr ähnlich zum binären Kleene-Stern mit dem Unterschied, dass der erste Ausdruck genau einmal akzeptieren muss, während der zweite Ausdruck unendlich oft akzeptieren muss.

2.2.2. Semantik

Die Semantik ist definiert als Relation $\models_{\text{ORE}}$ zwischen einem punktierten Wort und ω-regulären Ausdrücken. Die Relation $\models_{\text{ORE}}$ wird induktiv in Tabelle 2.2 definiert, wobei a und b ω-reguläre Ausdrücke sind, x und y reguläre Ausdrücke, w ein Wort und i eine Position.

Relation	Semantik
$(w, i) \models_{\text{ORE}} x; y^\omega$	Es existiert ein k, sodass $(w, i, k) \models_{\text{RE}} x$ gilt, und es existiert eine unendliche Sequenz $(i_0, i_1, \ldots)$ mit $i_0 = k$, sodass für alle l, $(w, i_l, i_{l+1}) \models_{\text{RE}} y$ gilt.
$(w, i) \models_{\text{ORE}} a \| b$	Entweder gilt $(w, i) \models_{\text{ORE}} a$ oder $(w, i) \models_{\text{ORE}} b$ oder beides.

Tabelle 2.2.: Induktive Definition der Relation $\models_{\text{ORE}}$ zur Beschreibung der Semantik von ω-regulären Ausdrücken. Die Relation gilt genau dann, wenn die angegebene Semantik gilt.

Dabei ist hervorzuheben, dass bei $x; y^\omega$ weder x noch y eine Sprache definieren können, die das leere Wort enthält. Auf diese Weise wird sichergestellt, dass eine ω-reguläre Sprache definiert wird. Für den Fall, dass der endliche Teil x leer sein soll, kann für einen regulären Ausdruck y folgende Äquivalenz benutzt werden

$$yy; y^\omega \equiv y; y^\omega.$$

Ein ω-regulärer Ausdruck x definiert die Sprache $\mathcal{L}(x)$, indem für ein Wort $w \in \Sigma^\omega$, $w \in \mathcal{L}(x)$ genau dann gilt, wenn $(w, 0) \models_{\text{ORE}} x$.

2.3. Reguläre Linearzeit-Temporallogik (RLTL)

Die reguläre Linearzeit-Temporallogik (RLTL) ist eine Alternative gleicher Mächtigkeit zu ω-regulären Ausdrücken. Von der Struktur her kann RLTL als Verallgemeinerung von LTL und ω-regulären Ausdrücken verstanden werden. Durch die Power-Operatoren können alle Operatoren aus LTL und die Idee der unendlichen Wiederholung einer durch einen regulären Ausdruck definierten Sprache kombiniert werden. Auf diese Weise steht wie in LTL die Möglichkeit der einfachen Negation und Konjunktion in einer Logik mit der Mächtigkeit von ω-regulären Sprachen zur Verfügung. Wir werden zudem sehen, wie ω-reguläre Ausdrücke und LTL-Formeln direkt und ohne Vergrößerung der Anzahl der Basisausdrücke in RLTL-Ausdrücke umgewandelt werden können.

2.3.1. Syntax

Die folgende Grammatik in EBNF definiert die Syntax von RLTL für unendliche Worte:

```
RLTL = B { "∨" B }          // Disjunktion
   B = O { "∧" O }          // Konjunktion
   O = E "/" RE "⟩⟩" E      // Power-Operator und ...
     | E "/" RE "⟩" E | E   // ... schwacher Power-Operatoren
   E = RE ";" E | C         // Konkatenation
   C = "∅"                  // leere Sprache, ...
     | "¬" C                // ... Negation und ...
     | "(" RLTL ")"         // ... Klammerung
```

Obige Grammatik von RLTL ist rein algebraisch und benötigt keine Fixpunktoperatoren, auch wenn wir später sehen werden, dass die Power-Operatoren durchaus auch als Fixpunkt-Gleichung definiert werden könnten.

Durch die Verwendung von RE als Grammatik der regulären Ausdrücke mit Vergangenheit erhalten wir direkt RLTL mit Vergangenheit, sodass RLTL keine eigenen Operationen in die Vergangenheit benötigt. Ebenso enthält die Syntax keine Basisausdrücke oder andere Elemente eines Alphabets, da diese durch die Verwendung der regulären Ausdrücke aufgenommen werden. Die Operatoren $\wedge$, $\vee$ und $\neg$ sind als klassische Konjunktion, Disjunktion und Negation zu verstehen. Informell beschreibt $\cdot;\cdot$ die Konkatenation eines regulären Ausdrucks mit einem RLTL-Ausdruck. Ersterer

beschreibt eine endliche Sprache, letzterer eine unendliche Sprache. $\emptyset$ repräsentiert die leere Sprache und akzeptiert kein Wort. Entsprechend akzeptiert $\neg\emptyset$ alles und kann damit in dem Ausdruck $x;\neg\emptyset$ einen regulären Ausdruck x, der eine endliche Sprache beschreibt, in die Unendlichkeit heben. Wird das endliche Präfix eines Wortes vom regulären Ausdruck akzeptiert, so wird das ganze Wort akzeptiert.

RLTL basiert auf dem Power-Operatoren $\cdot/\cdot\rangle\cdot$ und seinem schwachen Äquivalent $\cdot/\cdot\rangle\cdot$. Diese vereinen die Ideen verschiedener anderer Logiken in RLTL und wir werden sehen, dass durch diese Operatoren ω-reguläre Ausdrücke und LTL-Ausdrücke sehr leicht in RLTL-Ausdrücke umgewandelt werden können. Beide Power-Operatoren sind über drei Operanden definiert:

– Einen RLTL-Ausdruck, die *Pflicht*,
– einen regulären Ausdruck, die *Verzögerung*,
– und einen weiteren RLTL-Ausdruck, den *Versuch*.

Das Prinzip besteht nun darin, dass mit jedem Lesen der Verzögerung die Pflicht gelten muss, bis der Versuch gilt und damit die Wiederholung beendet. Der Unterschied zwischen dem starken und dem schwachen Power-Operator besteht darin, dass beim starken Power-Operator nach endlicher Wiederholung der Versuch erfüllt sein muss, während beim schwachen Power-Operator die Wiederholung auch unendlich lange vorkommen darf und der Versuch damit nie gültig wird. Ganz ähnlich kann auch der LTL-Ausdruck $x\,\mathcal{U}\,y$ als Kombination des Versuchs y und der Pflicht x mit einer impliziten Verzögerung von genau einem beliebigen Zeichen verstanden werden.

2.3.2. Semantik

Die Semantik ist wie bei ω-regulären Ausdrücken als Relation $\models_{\text{RLTL}}$ zwischen einem punktierten Wort und RLTL-Ausdrücken definiert. Die Relation $\models_{\text{RLTL}}$ wird induktiv in Tabelle 2.3 auf der nächsten Seite definiert, wobei a und b RLTL-Ausdrücke sind, x ein regulärer Ausdruck, w ein Wort und i eine Position.

Mit dieser Semantik lässt sich zum Beispiel die Sprache, in der jedes zweite Zeichen p erfüllt, durch $p;\neg\emptyset/\operatorname{true}\operatorname{true}\rangle\emptyset$ ausdrücken. Diese Sprache kann in LTL nicht definiert werden, da es keine Möglichkeit gibt, global nur für jedes zweite Zeichen Bedingungen aufzustellen. Als ω-regulärer Ausdruck kann diese Sprache trivial als $\operatorname{true};(p\operatorname{true})^\omega$ definiert werden, wenn es sich bei p um einen Basisausdruck bzw. eine Proposition handelt. Dabei ist true wieder als spezielles Zeichen des Eingabealphabets zu verstehen, dass für eine beliebige Proposition steht. Bei komplizierten Ausdrücken, insbesondere bei Negationen können ω-reguläre Ausdrücke jedoch sehr schnell kompliziert und unlesbar werden.

Relation	Semantik
$(w,i) \models_{\text{RLTL}} \emptyset$	Nie.
$(w,i) \models_{\text{RLTL}} a \vee b$	Entweder gilt $(w,i) \models_{\text{RLTL}} a$ oder $(w,i) \models_{\text{RLTL}} b$ oder beides.
$(w,i) \models_{\text{RLTL}} a \wedge b$	Es gilt $(w,i) \models_{\text{RLTL}} a$ und $(w,i) \models_{\text{RLTL}} b$.
$(w,i) \models_{\text{RLTL}} \neg a$	Es gilt $(w,i) \not\models_{\text{RLTL}} a$.
$(w,i) \models_{\text{RLTL}} x;a$	Es existiert eine Position k, sodass $(w,i,k) \models_{\text{RE}} x$ und $(w,k) \models_{\text{RLTL}} a$ gelten.
$(w,i) \models_{\text{RLTL}} a/x\rangle\rangle b$	Entweder 1) es gilt $(w,i) \models_{\text{RLTL}} b$ oder 2) es existiert eine endliche Folge $(i_0,i_1,\ldots,i_m)$ mit $i_0 = i$, sodass für alle $k < m$ die Beziehungen $(w,i_k,i_{k+1}) \models_{\text{RE}} x$ und $(w,i_k) \models_{\text{RLTL}} a$ gelten und schließlich $(w,i_m) \models_{\text{RLTL}} b$ gilt.
$(w,i) \models_{\text{RLTL}} a/x\rangle b$	Entweder 1) es gilt $(w,i) \models_{\text{RLTL}} a/x\rangle\rangle b$ oder 2) es existiert eine unendliche Folge $(i_0,i_1,\ldots)$ mit $i_0 = i$, sodass für alle k die Beziehungen $(w,i_k,i_{k+1}) \models_{\text{RE}} x$ und $(w,i_k) \models_{\text{RLTL}} a$ gelten.

Tabelle 2.3.: Induktive Definition der Relation $\models_{\text{RLTL}}$ zur Beschreibung der Semantik von RLTL. Die Relation gilt genau dann, wenn die angegebene Semantik gilt.

Weiter wird im Folgenden der Ausdruck

$$\text{first} :\equiv \neg(\text{notfirst}; \neg\emptyset)$$

sehr nützlich werden, um zu bestimmen, ob der Anfang eines Wortes erreicht ist.

Schließlich sei an dieser Stelle noch bemerkt, dass die Power-Operatoren als Fixpunkte aufgefasst werden können, auch wenn zur Definition von RLTL keine Fixpunkte verwendet wurden. So kann der Power-Operator $a/x\rangle\rangle b$ als kleinster Fixpunkt und der schwache Power-Operator $a/x\rangle b$ als größter Fixpunkt der Funktion $f : 2^{\Sigma^\omega} \to 2^{\Sigma^\omega}$ mit

$$f(X) = b \vee (a \wedge x; X)$$

verstanden werden.

2.3.3. Umwandlung von ω-regulären Ausdrücken in RLTL

Durch die im Folgenden induktiv definierte Funktion f kann jeder ω-reguläre Ausdruck in einen RLTL-Ausdruck umgewandelt werden. Dabei bleibt die Größe des Ausdrucks in dem Sinne erhalten, als jeder Teil der Formel direkt umgewandelt

werden kann und dabei nicht verdoppelt werden muss. Für beliebige ω-reguläre Ausdrücke a und b und reguläre Ausdrücke x und y gelte

$$f(a\|b) = f(a) \vee f(b)$$
$$f(x; y^\omega) = x; (\neg\emptyset/y\rangle\emptyset).$$

Die unendliche Wiederholung durch den ω-Operator wird also repräsentiert durch einen schwachen Power-Operator, mit Pflicht y, der Verzögerung eines beliebigen Zeichens und dem Versuch der leeren Sprachen. Da die leere Sprache kein punktiertes Wort akzeptiert, muss y unendlich oft gelten, da der Versuch keine Worte akzeptiert.

2.3.4. Entfernung von Negationen in RLTL

Durch die Einführung weiterer dualer Operatoren zu dem Konkatenations-Operator und beiden Power-Operatoren kann der Negations-Operator iterativ in die jeweiligen Operatoren hineingeschoben werden, bis eine positive Normalform entsteht. Eine positive Normalform ist dabei eine Formel, in der die Negation nur im Ausdruck $\neg\emptyset$ vorkommt, der neben $\emptyset$ als eigene Konstante angesehen werden kann. Diese Auffassung ist auch vor dem Hintergrund sinnvoll, dass sowohl zu $\emptyset$ als auch zu $\neg\emptyset$ entsprechende Automaten existieren, von denen jede Umwandlung ausgeht. Für die neuen Operatoren sollen die folgenden Äquivalenzen gelten

$$\neg(x; a) \equiv x; ; \neg a$$
$$\neg(x; ; a) \equiv x; \neg a$$
$$\neg(a/x\rangle\rangle b) \equiv \neg a//x\rangle\neg b$$
$$\neg(a/x\rangle b) \equiv \neg a//x\rangle\rangle\neg b$$
$$\neg(a//x\rangle\rangle b) \equiv \neg a/x\rangle\neg b$$
$$\neg(a//x\rangle b) \equiv \neg a/x\rangle\rangle\neg b$$

Der Operator ; ; ist dabei als dualer Operator zum normalen Konkatenations-Operator ; zu verstehen. Akzeptiert dieser ein Wort, wenn für dieses mindestens eine Aufteilung existiert, so dass das endliche Präfix vom ersten Ausdruck akzeptiert wird und der Rest vom zweiten Ausdruck, so akzeptiert der duale Operator ein Wort, wenn der zweite Ausdruck den Rest für jede mögliche Aufteilung akzeptiert, bei der der erste Ausdruck das Präfix akzeptiert. Entsprechend sind die beiden neuen Power-Operatoren als duale Operatoren zu den normalen Power-Operatoren zu sehen. Die Semantik hinter den dualen Power-Operatoren erschließt sich am ehesten, wenn man obige Äquivalenzen betrachtet. Dabei ist zu beachten, dass nur die beiden RLTL-Ausdrücke negiert werden, während der reguläre Ausdruck erhalten bleibt. Auf diese

Weise wird die formale Semantik etwas unübersichtlich, aber es wird das Problem umgangen, dass reguläre Ausdrücke nicht trivial negiert werden können. Zusammen mit den de Morganschen Regeln und der doppelten Negation

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$
$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$
$$\neg\neg a \equiv a$$

ist mit diesen Äquivalenzen die vollständige Entfernung des Negations-Operators möglich.

Dazu definieren wir in Tabelle 2.4 die formale Semantik der drei neuen Operatoren, wobei a und b RLTL-Ausdrücke sind, x ein regulärer Ausdruck, w ein Wort und i eine Position.

Relation	Semantik
$(w,i) \models_{\text{RLTL}} x;;a$	Für alle Positionen k mit $(w,i,k) \models_{\text{RE}} x$ gilt $(w,k) \models_{\text{RLTL}} a$.
$(w,i) \models_{\text{RLTL}} a//x\rangle\rangle b$	1) Es gilt $(w,i) \models_{\text{RLTL}} b$ und es gilt sowohl 2) für alle endlichen Folgen $(i_0,i_1,\ldots,i_m)$ mit $i_0 = i$, bei denen für alle $k < m$ gerade $(w,i_k,i_{k+1}) \models_{\text{RE}} x$ gilt, entweder $(w,i_j) \models_{\text{RLTL}} a$ für ein $j \leq m$ oder $(w,i_m) \models_{\text{RLTL}} b$ als auch 3) für alle unendlichen Folgen $(i_0,i_1,\ldots)$ mit $i_0 = i$, bei denen für alle k gerade $(w,i_k,i_{k+1}) \models_{\text{RE}} x$ gilt, ein m existiert mit $(w,i_m) \models a$.
$(w,i) \models_{\text{RLTL}} a//x\rangle b$	Entweder es gilt 1) $(w,i) \models_{\text{RLTL}} a//x\rangle\rangle b$ oder es gilt 2) $(w,i) \models_{\text{RLTL}} b$ und für alle unendlichen Folgen $(i_0,i_1,\ldots)$ mit $i_0 = i$, bei denen für alle k gerade $(w,i_k,i_{k+1}) \models_{\text{RE}} x$ gilt, für alle k gerade $(w,i_k) \models_{\text{RLTL}} b$ gilt.

Tabelle 2.4.: Erweiterung der induktiven Definition der Relation $\models_{\text{RLTL}}$ zur Beschreibung der Semantik von RLTL um drei neuen Operatoren. Die Relation gilt genau dann, wenn die angegebene Semantik gilt.

Analog zu den bereits eingeführten Power-Operatoren können auch die beiden dualen Power-Operatoren über eine Fixpunktiteration definiert werden. Dabei kann der duale Power-Operator $a//x\rangle\rangle b$ als kleinster und der schwache duale Power-Operator $a//x\rangle b$ als größter Fixpunkt der Funktion $f : 2^{\Sigma^\omega} \to 2^{\Sigma^\omega}$ mit

$$f(X) = a \wedge (b \vee x;;X)$$

aufgefasst werden.

2.3.5. Vollständige Syntax

Unter Hinzunahme der drei oben neu eingeführten dualen Operatoren ergibt sich folgende vollständige Grammatik für RLTL:

```
RLTL = B { "∨" B }              // Disjunktion
   B = O { "∧" O }              // Konjunktion
   O = E "/" RE "⟩⟩" E          // Power-Operator und ...
     | E "/" RE "⟩" E           // ... schwacher Power-Operatoren
     | E "//" RE "⟩⟩" E         // ... dualer Power-Operator und ...
     | E "//" RE "⟩" E | E      // ... schwacher dualer Power-Operator
   E = RE ";" E                 // Konkatenation
     | RE ";;" E | C            // ... duale Konkatenation
   C = "∅"                      // leere Sprache, ...
     | "¬" C                    // ... Negation und ...
     | "(" RLTL ")"             // ... Klammerung
```

Dabei ist `RE` wieder ein regulärer Ausdruck mit Vergangenheit wie oben definiert.

2.4. Linearzeit-Temporallogik (LTL) mit Vergangenheit

LTL wird üblicherweise als Beschreibung einer Menge von Wegen auf unendlichen, nicht verzweigten Pfaden verstanden. Diese Pfade können aber auch als unendliche Worte interpretiert werden. LTL kann dabei aber nicht alle ω-regulären Sprachen beschreiben. Zum Beispiel kann die bereits erwähnte Sprache, bei der jedes zweite Zeichen eine Bedingung erfüllt, nicht in LTL beschrieben werden.

In diesem Abschnitt wird LTL mit Vergangenheit eingeführt und gezeigt, wie diese LTL-Ausdrücke in RLTL-Ausdrücke umgewandelt werden können. Dabei werden neben den für die volle Mächtigkeit von LTL nötigen Operatoren zusätzlich weitere Operatoren über Äquivalenzen definiert, um zu demonstrieren, dass alle gängigen LTL-Operatoren direkt und unter Beibehaltung der Anzahl Basisausdrücke in RLTL-Ausdrücke umgewandelt werden können.

2.4.1. Syntax

Die folgende Grammatik in EBNF definiert die Syntax von LTL für unendliche Worte:

```
LTL = C { "∨" C }                                  // Disjunktion
  C = I { "∧" I }                                  // Konjunktion
  I = B [ "→" B ]                                  // Implikation
  B = U "𝒰" B | U "𝒲" B | U "ℛ" B                  // binaere Operatoren
    | U "𝒮" B | U "ℬ" B | U "𝒯" B | U
  U = "○" U | "◇" U | "□" U                        // unaere Operatoren, ...
    | "⊖" U | "⊟" U | "⟐" U | "⊡" U
    | "¬" U                                        // ... Negation und ...
    | P | "(" LTL ")"                              // ... Klammerung
```

Dabei ist `P` ganz analog zur Verwendung bei den regulären Ausdrücken als Basisausdruck zu verstehen, der true, false oder ein beliebiges anderes Zeichen des Eingabealphabets sein kann.

Die Operatoren der Disjunktion ($\vee$), Konjunktion ($\wedge$), Negation ($\neg$) und Implikation ($\rightarrow$) haben die klassische Bedeutung. Ebenso haben die Operatoren Until ($\mathcal{U}$), Release ($\mathcal{R}$), Next ($\bigcirc$), Globally ($\Box$) und Finally ($\Diamond$) die aus den meisten Verwendungen von LTL bekannte Semantik.

Weak-Until ($a \,\mathcal{W}\, b$) ist als schwacher Until-Operator zu verstehen, der ein Wort auch dann akzeptiert, wenn unendlich lange a gilt und b nie erreicht wird.

Damit auch Ausdrücke mit Vergangenheit gebildet werden können, existieren zu allen Ausdrücken, mit denen Aussagen über die Zukunft getroffen werden können analoge Operatoren für die Vergangenheit. Der Previous-Operator ($\ominus$) akzeptiert ein Wort an einer Position analog zum Next-Operator, wenn das vorherige Zeichen die Formel erfüllt. Der Previous-Operator ($\ominus$) verlangt dabei, dass der Schritt nach links in die Vergangenheit möglich ist, während der Weak-Previous-Operator ($\circledcirc$) direkt am Anfang des Wortes alles akzeptiert. Der Since-Operator ($\mathcal{S}$) entspricht einem Until in die Vergangenheit, der Back-Operator ($\mathcal{B}$) entspricht einem Weak-Until in die Vergangenheit und der duale Trigger-Operator ($\mathcal{T}$) entspricht einem Release in die Vergangenheit. Past-Globally ($\boxminus$) und Past-Finally ($\blacklozenge$) entsprechen schließlich Globally und Finally in die Vergangenheit.

2.4.2. Semantik

Die formale Semantik dieser Operatoren wird definiert, indem zunächst ein minimaler Satz an Operatoren als Relation $\models_{\text{LTL}}$ zwischen einem punktierten Wort und LTL-Ausdrücken definiert wird. Die Semantik der übrigen Operatoren wird dann als syntaktische Vereinfachung über Äquivalenzen ergänzt.

Die Relation $\models_{\text{LTL}}$ wird induktiv in Tabelle 2.5 auf der nächsten Seite definiert, wobei a und b LTL-Ausdrücke, w ein Wort und i eine Position sei.

Relation	Semantik
$(w,i) \models_{\text{LTL}} p$	$w[i]$ erfüllt p
$(w,i) \models_{\text{LTL}} a \vee b$	Entweder es gilt $(w,i) \models_{\text{LTL}} a$ oder $(w,i) \models_{\text{LTL}} b$ oder beides.
$(w,i) \models_{\text{LTL}} \bigcirc a$	Es gilt $(w,i+1) \models_{\text{LTL}} a$.
$(w,i) \models_{\text{LTL}} a \,\mathcal{U}\, b$	Es existiert ein $j \geq i$ mit $(w,j) \models_{\text{LTL}} b$ und für alle k mit $i \leq k < j$ gilt $(w,k) \models_{\text{LTL}} a$.
$(w,i) \models_{\text{LTL}} \ominus a$	Es gilt $i > 0$ und $(w,i-1) \models_{\text{LTL}} a$.
$(w,i) \models_{\text{LTL}} a \,\mathcal{S}\, b$	Es existiert ein $j \leq i$ mit $(w,k) \models_{\text{LTL}} b$ und für alle k mit $j < k \leq i$ gilt $(w,k) \models_{\text{LTL}} a$.

Tabelle 2.5.: Indutkive Definition der Relation $\models_{\text{LTL}}$ zur Beschreibung der Semantik von LTL. Die Relation gilt genau dann, wenn die angegebene Semantik gilt.

Die Konjunktion kann klassisch durch

$$a \wedge b :\equiv \neg(\neg a \vee \neg b)$$

definiert werden. Weiter gelten die folgenden Äquivalenzen für Operatoren in die Zukunft

$$\begin{aligned} \Diamond a &:\equiv \text{true}\,\mathcal{U}\, a \\ \Box a &:\equiv \neg \Diamond \neg a \\ a \,\mathcal{R}\, b &:\equiv \neg(\neg a \,\mathcal{U}\, \neg b) \\ a \,\mathcal{W}\, b &:\equiv (a \,\mathcal{U}\, b) \vee \Box a \end{aligned}$$

und folgende Äquivalenzen für Operatoren in die Vergangenheit

$$\begin{aligned} \boxminus a &:\equiv a \,\mathcal{B}\, \text{false} \\ \diamondminus a &:\equiv \neg \boxminus \neg a \\ \circledcirc a &:\equiv \neg \ominus \neg a \\ a \,\mathcal{T}\, b &:\equiv \neg(\neg a \,\mathcal{S}\, \neg b) \\ a \,\mathcal{B}\, b &:\equiv (a \,\mathcal{S}\, b) \vee \boxminus a. \end{aligned}$$

Die Operatoren der Vergangenheit können die Mächtigkeit von LTL nicht ändern. Analog zu (nicht schreibenden) Zwei-Wege-Automaten kann man durch Umwandeln der LTL-Formel »vorher nachsehen«, statt in die Vergangenheit zurückzugehen. Dadurch erhöht sich die Anzahl der benötigten Basisausdrücke in der Formel jedoch ebenfalls analog zu zusätzlichen Zuständen in Automaten und wird unübersichtlicher. So kann zum Beispiel die Aussage, dass ein Alarm a einen Einbruch b unmittelbar vorher impliziert, so ausdrücken:

$$\Box(a \rightarrow \ominus b)$$

Ohne Vergangenheit könnte man auch weniger direkt

$$\neg a \wedge \Box(b \vee \bigcirc \neg a)$$

schreiben.

2.5. Umwandlung von LTL zu RLTL

Ein LTL-Ausdruck kann direkt in einen RLTL-Ausdruck umgewandelt werden, sodass der RLTL-Ausdruck die gleiche Anzahl Basisausdrücke wie die ursprüngliche LTL-Formeln hat. Dazu definieren wir induktiv eine Funktion $f : \texttt{LTL} \to \texttt{RLTL}$ für einen Basisausdruck p und LTL-Ausdrücke a und b.

$$\begin{aligned}
f(p) &:= p; \neg\emptyset \\
f(a \wedge b) &= f(a) \wedge f(b) \\
f(a \vee b) &= f(a) \vee f(b) \\
f(\neg\, a) &= \neg f(a) \\
f(\bigcirc\, a) &= \text{true}; f(a) \\
f(a \,\mathcal{U}\, b) &= f(a) / \text{true}\rangle\rangle f(b) \\
f(\ominus\, a) &= -\,\text{true}; f(a) \\
f(a \,\mathcal{S}\, b) &= f(a) / -\,\text{true}\rangle\rangle f(b) \\
f(\Diamond\, a) &= \neg\emptyset / \text{true}\rangle\rangle f(a) \\
f(\Box\, a) &= f(a) / \text{true}\rangle\emptyset \\
f(\boxminus\, a) &= f(a) / -\,\text{true}\rangle\rangle\, \text{first} \\
f(\diamondminus\, a) &= \neg\emptyset / -\,\text{true}\rangle\rangle f(a) \\
f(a \,\mathcal{R}\, b) &= f(a) // \text{true}\rangle f(b) \\
f(a \,\mathcal{T}\, b) &= f(a) // -\,\text{true}\rangle(f(b) \vee \text{first}) \\
f(a \,\mathcal{W}\, b) &= f(a) / \text{true}\rangle f(b) \\
f(a \,\mathcal{B}\, b) &= f(a) / -\,\text{true}\rangle\rangle(f(a) \vee \text{first}) \\
f(\circledcirc\, a) &= \text{first} \vee -\,\text{true}; f(a)
\end{aligned}$$

Die Funktion f ist durch obige Definitionen wohldefiniert. Jedem LTL-Ausdruck wird eindeutig ein RLTL-Ausdruck zugeordnet. Umgekehrt lässt sich aber mit dieser Zuordnung nicht jeder RLTL-Ausdruck in einen LTL-Ausdruck umformen, da insbesondere als Verzögerungsoperand des Power-Operators in dieser Umwandlung nur die Spezialfälle true und $-\,\text{true}$ benötigt werden.

Da die Relationen zur Definition der Semantik von LTL und RLTL jeweils auf punktierten Worten definiert sind, können wir definieren: Ein RLTL-Ausdruck a und ein LTL-Ausdruck b sind genau dann äquivalent, wenn für alle Worte $w \in \Sigma^\omega$ gilt

$$(w, 0) \models_{\mathtt{RLTL}} a \iff (w, 0) \models_{\mathtt{LTL}} b.$$

Damit ist die Äquivalenz zwischen einem LTL- und einem RLTL-Ausdruck als Äquivalenz der definierten Mengen punktierter Worte zu verstehen. Oben definierte Umwandlung liefert zu jedem LTL-Ausdruck einen in diesem Sinne äquivalenten RLTL-Ausdruck.

3. Automaten

Bei der Umwandlung von RLTL-Ausdrücken in Paritätsautomaten werden zunächst die vorkommenden regulären Ausdrücke in nichtdeterministische endliche Automaten umgewandelt und diese entsprechend des RLTL-Ausdrucks zu alternierenden Paritätsautomaten zusammengesetzt. Enthalten die regulären Ausdrücke Operatoren der Vergangenheit, so entstehen nichtdeterministische endliche Zwei-Wege-Automaten, die zu einem alternierenden Zwei-Wege-Paritätsautomaten zusammengesetzt werden. Wir werden daher in diesem Kapitel die benötigten Automaten definieren.

3.1. Automaten auf endlichen Worten

Wir definieren zunächst nichtdeterministische endliche Automaten (NFA) und erweitern diese dann um ε-Transitionen. Basierend auf NFAs führen wir dann universelle Automaten (UFA) ein und ergänzen schließlich UFA und NFA zu Zwei-Wege-Automaten.

3.1.1. Nichtdeterministische endliche Automaten (NFA)

Nichtdeterministische endliche Automaten akzeptieren endliche Worte und definieren so endliche Sprachen. Sie haben damit die gleiche Mächtigkeit wie reguläre Ausdrücke.

Syntax

Ein nichtdeterministischer endlicher Automat (NFA) besteht aus einem 5-Tupel $(\Sigma, Q, Q_0, \Delta, F)$. Dabei ist

- Σ das Eingabealphabet,
- Q die Menge der Zustände,
- $Q_0 \subseteq Q$ die Menge der Startzustände,
- $\Delta \subseteq (Q \times \Sigma) \times Q$ die Transitionsrelation und
- $F \subseteq Q$ die Menge der akzeptierenden Zustände.

Dieses Tupel kann zur besseren Übersichtlichkeit als Graph dargestellt werden. Dabei werden die Zustände als Knoten dargestellt und es gibt eine gerichtete Kante mit Beschriftung a von q nach q', wenn $((q, a), q') \in \Delta$ gilt. Akzeptierende Zustände werden durch einen Doppelkreis um den Zustand dargestellt und Startzustände werden durch einen Pfeil vom Wort »start« zum entsprechenden Knoten markiert.

So kann zum Beispiel der Automat

$$
\begin{aligned}
&(\{a, b\}, && \textit{// Alphabet}\\
&\{q_1, q_2, q_3\}, && \textit{// Zustände}\\
&\{q_1, q_2\}, && \textit{// Startzustände}\\
&\{((q_1, a), q_2), ((q_2, a), q_1),\\
&\quad ((q_1, b), q_3), ((q_2, b), q_3)\}, && \textit{// Transitionen}\\
&\{q_3\}) && \textit{// akzeptierende Zustände}
\end{aligned}
$$

als Graph in Abbildung 3.1 dargestellt werden.

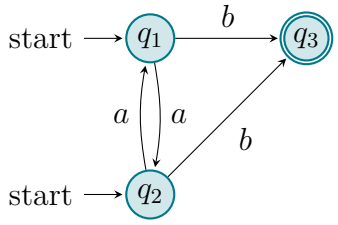

Abbildung 3.1.: Beispiel eines NFA in Graphendarstellung.

Semantik

Wir definieren NFAs analog zu regulären Ausdrücken auf Segmenten von unendlichen Worten. Eine Konfiguration eines NFA ist damit ein Tupel bestehend aus einem Zustand q des Automaten und einem Segment (w, i, j).

Ein Lauf eines NFAs besteht aus einer endlichen Folge von Konfigurationen, wobei zwei Konfigurationen $(q, (w, i, j))$ und $(q', (w, i', j))$ aufeinander folgen, wenn $i' = i + 1$ und $((q, w[i]), q') \in \Delta$ gilt. Ein akzeptierender Lauf für eine Eingabe (w, i, j) ist ein Lauf, der mit $(q_0, (w, i, j))$ für $q_0 \in Q$ beginnt und mit $(q, (w, i', j))$ für $i' = j$ und $q \in F$ endet. Im Gegensatz zum hier nicht weiter betrachteten deterministischen endlichen Automaten kann es für eine Eingabe (w, i, j) mehrere Läufe geben. Der NFA akzeptiert eine solche Eingabe, wenn dafür mindestens ein akzeptierender Lauf existiert.

Um die Transitionsrelation einfacher implementieren zu können, kann diese auch als Transitionsfunktion $\delta : Q \times \Sigma \to 2^Q$ interpretiert werden. Dabei gilt

$$\delta(q, a) := \{q' \mid ((q, a), q') \in \Delta\}.$$

3.1.2. Syntaktische Vereinfachungen

Soll ein Automat in einem Zustand auf alle gelesenen Zeichen gleich reagieren, muss für jedes Zeichen ein eigenes Element in der Transitionsrelation notiert werden. Dies kann vereinfacht werden, indem das Fragezeichen »?« als Platzhalter für ein beliebiges Zeichen zugelassen wird.

Bei einem NFA entspricht also $((q, ?), q') \in \Delta$ der Aussage

$$\forall a \in \Sigma \quad ((q, a), q') \in \Delta.$$

Für alle im Folgenden definierten Automaten ist »?« analog zu verstehen.

3.1.3. Totale Automaten

Unabhängig davon, ob eine Transitionsrelation oder eine Transitionsfunktion verwendet wird, kann es Kombinationen von einem Zustand q und einem gelesenem Zeichen a geben, für die keine Folgezustände angegeben sind. Es existiert also kein $q' \in Q$ mit $((q, a), q') \in \Delta$ bei Verwendung der Transitionsrelation bzw. es gilt $\delta(q, a) = \emptyset$ bei Verwendung der Transitionsfunktion. Wird trotzdem im Zustand q das Zeichen a gelesen, so kann dieser Lauf kein akzeptierender Lauf sein. Informell wird die Eingabe direkt verworfen, wenn für das aktuell gelesene Zeichen kein Folgezustand angegeben ist.

Kann dieser Fall nicht eintreten, ist also für jede mögliche Kombination von Zustand q und Zeichen a mindestens ein Folgezustand angegeben, so wird ein Automat als *totaler Automat* bezeichnet. Jeder Automat A kann durch Hinzufügen eines zusätzlichen Zustandes zu einem totalen Automaten A' ergänzt werden. Für jedes Zeichen des Eingabealphabets existiert eine Transitionsschleife von diesem neuen Zustand (im Folgenden *Senke* genannt) aus zu diesem Zustand zurück. Wurde dieser Zustand also einmal erreicht, kann er nicht mehr verlassen werden. Dieser Zustand ist nicht akzeptierend. In einem akzeptierenden Lauf darf dieser Zustand also nicht enthalten sein. Für jede Kombination von Zustand q und Eingabe a, für die im Automaten A kein Folgezustand angegeben ist, wird nun die neue Senke als Folgezustand angegeben. Auf diese Weise wird die Semantik nicht verändert, aber es existiert nun im neuen Automaten A' immer ein Folgezustand für jeden Zustand und jede gelesene Eingabe. Abbildung 3.2 auf Seite 33 zeigt eine solche Senke in der Darstellung als Graph.

3.1.4. ε-Transitionen

Wir wollen zusätzlich ε-Transitionen erlauben und erweitern die Transitionsrelation dazu auf

$$\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q.$$

Dabei gelte $\varepsilon \notin \Sigma$.

In einem Lauf eines NFAs mit ε-Transitionen folgen nun zwei Konfigurationen $(q, (w, i, j))$ und $(q', (w, i', j))$ aufeinander, wenn wie bisher $i' = i + 1$ gilt und $((q, w[i]), q') \in \Delta$ oder $i' = i$ und $((q, \varepsilon), q') \in \Delta$ gelten. Eine ε-Transition erlaubt also den Wechsel von Zuständen ohne ein Zeichen zu lesen. Das wird insbesondere für die Kombination von mehreren Automaten zu einem größeren Automaten in der Bottom-Up-Konstruktion der Umwandlung nützlich werden.

3.1.5. Universelle Automaten (UFA)

Ein universeller Automat wird genau wie ein NFA definiert. Der einzige Unterschied besteht darin, dass ein universeller Automat eine Eingabe (w, i, j) nur dann akzeptiert, wenn *alle* Läufe für diese Eingabe akzeptierend sind. Existieren in einem Zustand bei einem gelesenen Zeichen mehrere Folgezustände, müssen also alle Folgezustände eingenommen werden und die jeweilige Fortsetzung des Laufs mit allen Folgezuständen muss zu einem akzeptierenden Lauf führen.

3.1.6. Zwei-Wege-Varianten (2NFA, 2UFA)

Bei einem Zwei-Wege-NFA (2NFA) muss nach dem Lesen eines Zeichens nicht mehr zwingend das nächste Zeichen betrachtet werden, sondern es kann noch einmal das gleiche Zeichen oder das vorherige Zeichen betrachtet werden. Diese Form von Automaten wird insbesondere für die Übersetzung von regulären Automaten mit Vergangenheit benötigt.

Dazu wird die Transitionsrelation erweitert zu

$$\Delta \subseteq (Q \times \Sigma) \times \hat{Q}$$

mit $\hat{Q} = Q \times \{-1, 0, 1\}$.

In einem Lauf eines 2NFAs folgen zwei Konfigurationen $(q, (w, i, j))$ und $(q', (w, i', j))$ aufeinander, wenn $i' = i + k$ und $((q, w[i]), (q', k)) \in \Delta$ für $k \in \{-1, 0, 1\}$ gilt.

Ein akzeptierenden Lauf für eine Eingabe (w, i, j) ist auch hier ein Lauf, der mit $(q_0, (w, i, j))$ für $q_0 \in Q$ beginnt und mit $(q, (w, i', j))$ für $i' = j$ und $q \in F$ endet, wobei jetzt zusätzlich keine Konfiguration $(q, (w, i', j))$ vorkommen darf mit $i' < i$.

Bei einem 2NFA mit ε-Transitionen wird die Transitionsrelation zu

$$\Delta \subseteq ((Q \times (\Sigma \cup \{\varepsilon\})) \times \hat{Q}).$$

Zwei Konfigurationen $(q, (w, i, j))$ und $(q', (w, i', j))$ folgen nun aufeinander, wenn $i' = i + k$ und $((q, w[i]), (q', k)) \in \Delta$ oder $i' = i$ und $((q, \varepsilon), (q', 0)) \in \Delta$ gilt. Eine ε-Transition hat also immer die Bewegungsrichtung 0.

Ein Zwei-Wege-UFA funktioniert ganz analog zu einem 2NFA, wobei wieder gefordert wird, dass *alle* Läufe für eine Eingabe akzeptierend sind.

In der Darstellung von Zwei-Wege-Automaten als Graph existiert eine gerichtete Kante von q nach q', wenn $((q, a), (q', d)) \in \Delta$ gilt. Diese wird mit a/d beschriftet. So kann zum Beispiel der Automat

$$\begin{aligned}
&(\{a, b\}, && \textit{// Alphabet}\\
&\{q_1, q_2, q_3\}, && \textit{// Zustände}\\
&\{q_1, q_2\}, && \textit{// Startzustände}\\
&\{((q_1, a), (q_2, 1)), ((q_2, a), (q_1, -1)),\\
&\quad ((q_1, b), (q_3, 1)), ((q_2, b), (q_3, 1))\}, && \textit{// Transitionen}\\
&\{q_3\}) && \textit{// akzeptierende Zustände}
\end{aligned}$$

als Graph in Abbildung 3.3 auf Seite 33 dargestellt werden.

3.2. Automaten auf unendlichen Worten

In diesem Abschnitt betrachten wir Automaten auf unendlichen Worten. Im Gegensatz zu den Automaten auf endlichen Worten aus dem vorherigen Abschnitt ist die Eingabe jetzt nicht mehr ein Segment eines unendlichen Wortes, sondern ein punktiertes Wort. Damit ist ein Lauf jetzt eine *unendliche* Folge von Konfigurationen. Somit kann als Akzeptanzbedingung nicht mehr die letzte Konfiguration im Lauf des Eingabewortes verwendet werden.

3.2.1. Büchi-Automaten (NBW)

Büchi-Automaten (NBW) sind die einfachste Variante von endlichen Automaten auf unendlichen Worten, da hier die Akzeptanz-Bedingung am einfachsten zu überprüfen ist und der Leerheitstest somit leicht erfolgen kann. Büchi-Automaten akzeptieren unendliche Sprachen und haben gerade die Mächtigkeit von ω-regulären Ausdrücken.

Wir werden im Folgenden verschiedene Erweiterungen dieses Automaten bzw. seiner Akzeptanzbedingungen betrachten. Durch diese Ergänzungen wird es deutlich einfacher Automaten anzugeben, die komplexe Bedingungen überprüfen. Die Mächtigkeit der Automaten ändert sich durch diese Erweiterungen aber nicht, sodass alle hier vorgestellten Automaten auf unendlichen Worten die gleiche Mächtigkeit haben und damit für jede ω-reguläre Sprache ein Automat existiert, der diese akzeptiert.

Syntax

Ein Büchi-Automat (NBW) besteht aus einem 5-Tupel $(\Sigma, Q, Q_0, \Delta, F)$. Dabei ist

- Σ das Eingabealphabet,
- Q die Menge der Zustände,
- $Q_0 \subseteq Q$ die Menge der Startzustände,
- $\Delta \subseteq (Q \times \Sigma) \times Q$ die Transitionsrelation und
- $F \subseteq Q$ die Menge der akzeptierenden Zustände.

Dieses Tupel kann ganz analog zum NFA als Graph dargestellt werden.

Semantik

Eine Konfiguration eines NBWs ist ein Tupel bestehend aus einem Zustand q des Automaten und einem punktierten Wort (w, i). Wir definieren NBWs direkt auf unendlichen Worten, sodass die initialen Konfigurationen $(w, 0)$ als punktiertes Wort enthalten.

Ein Lauf eines NBWs besteht aus einer unendlichen Folge π von Konfigurationen, wobei zwei Konfigurationen $(q, (w, i))$ und $(q', (w, i'))$ aufeinander folgen, wenn $i' = i+1$ und $((q, w[i]), q') \in \Delta$. Es sei $\inf(\pi)$ die Menge der unendlich oft vorkommenden Zustände in der Folge π von Konfigurationen. Ein akzeptierender Lauf π für eine Eingabe w ist ein Lauf, der mit $(q_0, (w, 0))$ für $q_0 \in Q_0$ beginnt und $\inf(\pi) \cap F \neq \emptyset$ erfüllt. Der NBW akzeptiert eine solche Eingabe, wenn dafür mindestens ein akzeptierender Lauf existiert.

3.2.2. ε-Transitionen und Zwei-Wege-Varianten (2NBW)

Durch Erweiterung der Transitionsrelation zu

$$\Delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\})) \times \hat{Q}$$

mit $\hat{Q} = Q \times \{-1, 0, 1\}$ kann der NBW zu einem Zwei-Wege-Büchi-Automaten mit ε-Transitionen erweitert werden.

Zwei Konfigurationen $(q, (w, i))$ und $(q', (w, i'))$ folgen nun aufeinander, wenn

$$i' = i + k \text{ und } ((q, w[i]), (q', k)) \in \Delta$$

oder

$$i' = i \text{ und } ((q, \varepsilon), (q', 0)) \in \Delta$$

gilt. Ein akzeptierender Lauf darf zusätzlich explizit keine Konfiguration $(q, (w, i'))$ mit $i' < 0$ enthalten.

3.2.3. Paritätsautomaten

Ein Paritätsautomat unterscheidet sich vom Büchi-Automat nur in der Akzeptanzbedingung. Ein Paritätsautomat besteht also genau wie ein NBW aus einem 5-Tupel $(\Sigma, Q, Q_0, \Delta, F_P)$. Dabei unterscheidet sich dieses Tupel nur in F_P vom NBW: Die Funktion $F_P : Q \to \mathbb{N}$ mit $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ weist jedem Zustand eine Zahl zu. Diese Zahl $F_P(q)$ wird *Parität* oder *Farbe* von q genannt.

Ein akzeptierender Lauf π für eine Eingabe w ist nun ein Lauf, der mit $(q_0, (w, 0))$ für $q_0 \in Q_0$ beginnt und bei dem $\max\{F_P(q) \mid q \in \inf(\pi)\}$ gerade ist. Der NBW akzeptiert eine solche Eingabe, wenn dafür mindestens ein akzeptierender Lauf existiert.

In dieser Arbeit werden insbesondere Paritätsautomaten mit den Farben 0, 1 und 2 verwendet. APWs mit dieser Einschränkung entstehen bei der in dieser Arbeit implementierten Umwandlung von RLTL und nur solche Automaten können auch mit dem im Rahmen von [Sch11] implementierten Programm in NBWs umgewandelt werden.

Ein Paritätsautomat kann wie ein NBW als Graph dargestellt werden. Ein Unterschied besteht nur in der Visualisierung der Akzeptanzbedingung. So werden keine Knoten doppelt umrandet, sondern jeder Knoten wird mit dem zugehörigen Zustand und der Parität dieses Zustandes getrennt durch einen Schrägstrich beschriftet.

So kann zum Beispiel der Automat

$$
\begin{aligned}
&(\{a, b\}, && \textit{// Alphabet}\\
&\{q_1, q_2, q_3\}, && \textit{// Zustände}\\
&\{q_1, q_2\}, && \textit{// Startzustände}\\
&\{((q_1, a), q_2), ((q_2, a), q_1), ((q_1, b), q_3),\\
&\quad ((q_2, b), q_3), ((q_3, a), q_2)\}, && \textit{// Transitionen}\\
&\{(q_1, 1), (q_2, 1), (q_3, 2)\}) && \textit{// Paritäten}
\end{aligned}
$$

als Graph in Abbildung 3.4 auf der nächsten Seite dargestellt werden.

3.2.4. Weitere Akzeptanzbedingungen

Durch Anpassung der Akzeptanzbedingung und dem Element F des 5-Tupels entstehen weitere Automaten auf unendlichen Worten, die hier der Vollständigkeit halber kurz erwähnt werden.

Muller-Automat $F_M \subseteq 2^Q$ ist eine Menge von Zustandsmengen. Ein akzeptierender Lauf π für eine Eingabe w ist nun ein Lauf, der mit $(q_0, (w, 0))$ für $q_0 \in Q_0$ beginnt und bei dem $\inf(\pi) \in F_M$ gilt. Eine solche Eingabe wird akzeptiert, wenn dafür mindestens ein akzeptierender Lauf existiert.

Rabin-Automat $F_R \subseteq 2^Q \times 2^Q$ ist eine Menge von 2-Tupeln von Zustandsmengen. Ein akzeptierender Lauf π für eine Eingabe w ist nun ein Lauf, der mit $(q_0, (w, 0))$ für $q_0 \in Q_0$ beginnt und bei dem mindestens ein solches 2-Tupel (U_i, V_i) existiert, sodass gilt

$$\inf(\pi) \cap U_i \neq \emptyset \quad \text{und} \quad \inf(\pi) \cap V_i = \emptyset.$$

Eine solche Eingabe wird akzeptiert, wenn dafür mindestens ein akzeptierender Lauf existiert.

Ein Rabin-Automat ist ein Spezialfall des Muller-Automaten. Formal lässt sich aus

$$F_R = \{(V_1, U_1), (V_2, U_2), \ldots (V_n, U_n)\}$$

die Akzeptanzbedingung eines Muller-Automaten herleiten:

$$F_M = \{M \mid M \subseteq Q \wedge \exists (U, V) \in F_R : M \cap U \neq \emptyset \wedge M \cap V = \emptyset\}.$$

Streett-Automat $F_S \subseteq 2^Q \times 2^Q$ ist eine Menge von 2-Tupeln von Zustandsmengen. Ein akzeptierender Lauf π für eine Eingabe w ist nun ein Lauf, der mit $(q_0, (w, 0))$ für $q_0 \in Q_0$ beginnt und bei dem für alle solchen 2-Tupel $(G_i, H_i) \in F_S$ gilt

$$\inf(\pi) \cap G_i \neq \emptyset \Rightarrow \inf(\pi) \cap H_i \neq \emptyset.$$

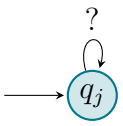

Abbildung 3.2.: Beispiel einer Senke q_j als Graph.

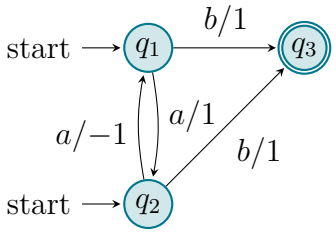

Abbildung 3.3.: Beispiel eines 2NFA in Graphendarstellung.

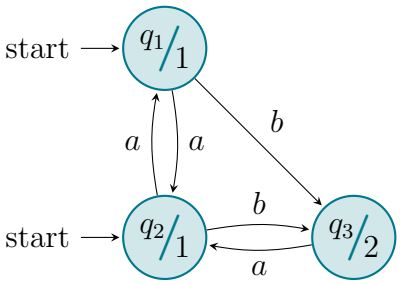

Abbildung 3.4.: Beispiel eines Paritätsautomaten in Graphendarstellung.

Eine solche Eingabe wird akzeptiert, wenn dafür mindestens ein akzeptierender Lauf existiert.

Ein Spezialfall ist der Streett-Automat mit nur einem Tupel $F_S = \{(G, H)\}$. Ein Paritätsautomat mit drei Farben kann auch als Streett-Automat mit nur einem Tupel verstanden werden:

$$F_S = \left\{\Big(\{q \mid q \in Q \wedge F_P(q) = 1\}, \{q \mid q \in Q \wedge F_P(q) = 2\}\Big)\right\}.$$

Bei einem Paritätsautomaten mit nur drei Farben sind alle Zustände der Farbe 0 neutral und wenn Zustände der Farbe 1 unendlich oft besucht werden, müssen auch Zustände der Farbe 2 unendlich oft besucht werden.

3.2.5. Alternierende Paritätsautomaten (APW)

Wir betrachten alternierende Automaten nur am Beispiel des alternierenden endlichen Paritätsautomaten auf unendlichen Wörtern (APW). Betrachtet man die Transitionsfunktion eines NBW, so bildet diese auf eine Menge von Zuständen ab. Diese Menge kann als eine Disjunktion von Zuständen interpretiert werden, da in jedem Lauf nur einer dieser Zustände als nächster Zustand besucht wird. Bei dieser Interpretation liegt die Idee nahe, auch Konjunktionen zuzulassen, sodass mehrere Folgezustände gleichzeitig besucht werden.

Anders als ein NBW befindet sich ein APW daher nicht zwingend nur in einer Konfiguration pro Leseschritt, sondern kann sich in mehreren Konfigurationen gleichzeitig befinden. Immer wenn eine Transition eine Verundung von mehreren Zuständen enthält, existieren mehrere Folge-Konfigurationen, wenn diese Transition benutzt wird. Alle diese Konfigurationen erzeugen im nächsten Leseschritt wieder mindestens eine Folge-Konfiguration und müssen nicht zwingend das gleiche punktierte Wort enthalten. Die Positionen im Eingabewort können also variieren.

Positive boolesche Formeln

Um boolesche Kombinationen von Zuständen formal zu erfassen, definieren wir $\mathcal{B}^+(Q)$ als die *positiven booleschen Formeln* über einer Menge von Propositionen Q. Eine solche Formel besteht aus true, false, den Elementen aus Q und den Operatoren $\vee$ und $\wedge$. Formal ergibt sich die Menge $\mathcal{B}^+(Q)$ über einer Grundmenge Q durch die folgende induktive Definition:

– Es gilt $\{\text{true}, \text{false}\} \subset \mathcal{B}^+(Q)$,
– $Q \subset \mathcal{B}^+(Q)$ und

– für alle $q, r \in \mathcal{B}^+(Q)$ gilt $q \vee r \in \mathcal{B}^+(Q)$ und $q \wedge r \in \mathcal{B}^+(Q)$.

$M \subseteq Q$ ist ein minimales Modell einer positiven booleschen Formel f über der Menge Q, wenn $M \models f$ gilt und kein $M' \subsetneq M$ existiert mit $M' \models f$. $M \models f$ ist dabei genau dann erfüllt, wenn die Formel f bei der Belegung aller Variablen aus M mit true, aller anderen Variablen mit false und der klassischen Semantik von $\wedge$ und $\vee$ zu true auswertet. Analog zu den Definitionen von LTL und RLTL bindet $\wedge$ dabei stärker als $\vee$.

Die Formel $f = a \wedge b \vee c$ ist zum Beispiel eine positive boolesche Formel über der Menge $\{a, b, c, d, e\}$ und die minimalen Modelle sind $\{c\}$ und $\{a, b\}$.

Schließlich sei auf zwei Spezialfälle hingewiesen: Für die Formel true ist das minimale Modell die leere Menge $\{\}$, da diese Formel immer zu true auswertet. Für die Formel false existiert kein minimales Modell, da diese Formel immer zu false auswertet.

Syntax

Ein alternierender endlicher Paritätsautomat auf unendlichen Wörtern (APW) besteht aus einem 5-Tupel $(\Sigma, Q, Q_0, \delta, F)$. Dabei ist

– Σ das Eingabealphabet,
– Q die Menge der Zustände,
– $Q_0 \in \mathcal{B}^+(Q)$ die initiale Zustandsformel,
– $\delta : (Q \times \Sigma) \rightsquigarrow \mathcal{B}^+(Q)$ die partielle Transitionsfunktion und
– $F : Q \to \mathbb{N}$ die Menge der akzeptierenden Zustände.

Eine partielle Funktion $f : A \rightsquigarrow B$ ist dabei eine rechtseindeutige Relation $f \subseteq A \times B$, die im Gegensatz zu einer Funktion nicht linkstotal sein muss. Die Transitionsfunktion δ ist eine partielle Funktion, da nicht für jede Kombination von Zustand und Eingabe eine positive boolesche Formel der nächsten Zustände definiert sein muss. Ist keine Formel angegeben, so existiert insbesondere natürlich auch kein minimales Modell. Ein APW kann also zum totalen Automaten (und damit die Transitionsfunktion zur totalen Funktion) erweitert werden, indem alle nicht angegebenen Transitionen als Transitionen zu false interpretiert werden.

Dieses Tupel kann analog zum Paritätsautomaten als Graph dargestellt werden. Dazu werden positive boolesche Formeln als Graph dargestellt. Dabei werden die Operatoren $\wedge$ und $\vee$ als Knoten in Rautenform dargestellt und die Formel als Baum bestehend aus diesen Knoten, den Zuständen und speziellen Knoten für true und false. Die Knoten für die Operatoren werden mit »$\wedge$« und »$\vee$« beschriftet. Der Knoten für true besteht aus einem auf der Grundseite stehenden Dreieck und der Beschriftung »true«, der Knoten für false besteht aus einem auf der Spitze stehenden Dreieck mit der Beschriftung »false«. Für jedes Vorkommen von $\wedge$ und $\vee$ in den

Formeln wird dem Graphen ein eigener Knoten hinzugefügt. Ist $f = \delta(q, a)$ für einen Zustand q definiert, so wird dem Graphen die Formel f und eine Kante vom Knoten q zum Wurzelknoten der Formel f hinzugefügt.

So kann zum Beispiel der Automat

$$
\begin{aligned}
&(\{a, b, c\}, && \textit{// Alphabet}\\
&\{q_0, q_1, q_2\}, && \textit{// Zustände}\\
&q_1 \wedge q_2, && \textit{// initiale Zustandsformel}\\
&\{((q_0, a), q_1 \wedge q_2 \vee \text{true}), ((q_2, b), q_0),\\
&\quad ((q_0, c), \text{false}), ((q_1, b), q_2)\}, && \textit{// Transitionen}\\
&\{(q_0, 0), (q_1, 0), (q_1, 1)\}) && \textit{// Paritäten}
\end{aligned}
$$

als Graph in Abbildung 3.5 dargestellt werden.

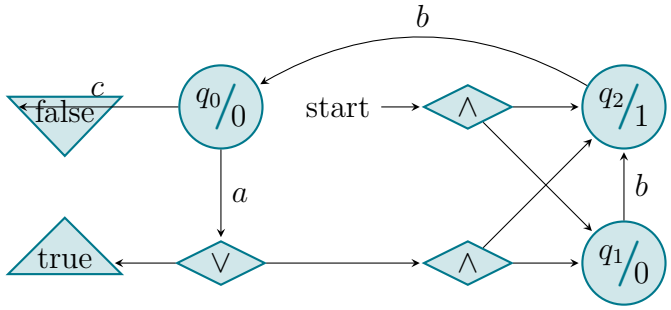

Abbildung 3.5.: Beispiel eines alternierenden Paritätsautomaten in der Darstellung als Graph.

Semantik

Ein Lauf eines APW besteht aus einem gerichteten azyklischen Graphen (DAG) $(V, E = V \times V)$, bei dem die Knoten $V \subseteq (Q, (\Sigma^\omega, \mathbb{N}))$ Konfigurationen des APW entsprechen. Eine Konfiguration $(q, (w, i))$ besteht dabei wie beim NBW aus einem Zustand q und einem punktierten Wort (w, i).

Ein Lauf für eine Eingabe w erfüllt nun die folgenden Bedingungen: Es gibt ein minimales Modell M_0 von Q_0, sodass für alle $m_0 \in M_0$, ein Knoten $(m_0, (w, 0))$ existiert. Weiter existiert für jeden Knoten $(q, (w, i))$ ein minimales Modell M von $\delta(q, w[i])$, sodass für alle Zustände $m \in M$ eine Kante von $(q, (w, i))$ zu $(m, (w, i+1))$ existiert.

Eine Spur in einem Lauf sei nun eine unendliche Folge von Konfigurationen beginnend mit der Wurzel des Laufs, sodass zwei Zustände aufeinander folgen, wenn sie

im Lauf durch eine gerichtete Kante verbunden sind. Eine Spur ist also ein Pfad durch den Lauf, der an der Wurzel des Laufs beginnt.

Es sei analog zum NBW $\inf(\pi)$ die Menge der unendlich oft besuchten Zustände in einer Spur π. Ein akzeptierender Lauf π für eine Eingabe w ist dann ein Lauf, bei dem alle Pfade die Akzeptanzbedingung des Paritätsautomaten erfüllen. Es muss also für jede Spur des Laufs $\max\{F(q) \mid q \in \inf(\pi)\}$ gerade sein. Existiert für eine Eingabe w mindestens ein akzeptierender Lauf, so akzeptiert der APW das Wort w.

Mit diesem Konzept von Läufen und Pfaden bilden alternierende Automaten eine Verallgemeinerung von nicht-deterministischen Automaten und universellen Automaten. Sobald in einer Formel $\delta(q, w[i])$ eine Disjunktion vorkommt, existieren mehrere minimale Modelle für diese Formel. In diesen Fällen existieren analog zu anderen nichtdeterministischen Automaten mehrere Läufe zu einer Eingabe, von denen nur einer akzeptierend sein muss. Sobald in einer Formel $\delta(q, w[i])$ eine Konjunktion vorkommt, existieren analog zu universellen Automaten mehrere Pfade zu einer Eingabe, die alle akzeptierend sein müssen. Für einen alternierenden Automaten, bei dem die minimalen Modelle alle höchstens ein Element haben, existieren viele Läufe, aber für jeden Lauf nur eine Spur. In diesem Fall bestehen die Transitionsformeln alle nur aus Disjunktionen. Ein solcher alternierender Automat entspricht einem nichtdeterministischen Automaten. Für einen alternierenden Automaten, bei dem es für jede Formel genau ein minimales Modell gibt, existiert nur ein Lauf, aber viele Spuren innerhalb dieses einen Laufs. Diese Bedingung lässt in den Formeln der Transitionsfunktion nur Konjunktionen zu. Ein solcher Automat entspricht einem universellen Automaten. Für einen alternierenden Automaten, bei dem für jede Formel genau ein minimales Modell bestehend aus genau einem Zustand existiert, existiert genau ein Lauf mit einer Spur. In einem solchen Automaten besteht jede Formel in der Transitionsfunktion aus genau einem Zustand. Ein solcher alternierender Automat entspricht schließlich einem deterministischen Automaten.

3.2.6. ε-Transitionen und Zwei-Wege-Varianten (2APW)

Durch Erweiterung der Transitionsfunktion zu

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightsquigarrow \mathcal{B}^+(\hat{Q})$$

mit $\hat{Q} = Q \times \{-1, 0, 1\}$ kann der APW zu einem Zwei-Wege-APW mit ε-Transitionen (2APW) erweitert werden.

Dabei ist zu beachten, dass die Transitionsfunktion nun auf $\mathcal{B}^+(\hat{Q})$ abbildet, während die initialen Zustände noch immer $Q_0 \in \mathcal{B}^+(Q)$ sind und somit keine Bewegungsrichtung erhalten.

Ein Lauf für eine Eingabe w erfüllt nun die folgenden Bedingungen: Es gibt ein minimales Modell M_0 von Q_0, sodass wie vorher für alle $m_0 \in M_0$, ein Knoten $(m_0, (w, 0))$ existiert. Weiter existiert zwischen zwei Knoten $(q, (w, i))$ und $(q', (w, i'))$ genau dann eine Kante in E, wenn 1) ein minimales Modell M von $\delta(q, w[i])$ existiert, sodass $(q', i' - i) \in M$ gilt oder 2) ein minimales Modell M von $\delta(q, \varepsilon)$ existiert, sodass $(q', 0) \in M$ gilt. Zusätzlich muss nun gefordert werden, dass es keinen Knoten $(q, (w, i)) \in V$ mit $i < 0$ gibt.

$\inf(\pi)$ wird auch hier als Menge aller unendlich oft vorkommenden Zustände einer Spur definiert und eine akzeptierende Spur kann nun wie bisher als Spur π für eine Eingabe w definiert werden, bei der $\max\{F(q) \mid q \in \inf(\pi)\}$ gerade ist. Existiert für eine Eingabe w mindestens ein Lauf, dessen Spuren alle akzeptierend sind, so akzeptiert der 2APW das Wort w.

Bei der Darstellung eines 2APW als Graph wird anders als beim Graphen eines 2NFA die ausgehende Kante an einem Zustand nur mit dem zu lesenden Zeichen beschriftet. Die eingehende Kante an einem Zustand wird mit einem Schrägstrich gefolgt von der Bewegungsrichtung (-1, 0 oder 1) beschriftet. Wenn für alle Zustände einer booleschen Formel die Bewegungsrichtung gleich ist, kann diese getrennt vom zu lesenden Zeichen durch einen Schrägstrich direkt an der am Zustand ausgehenden Kante notiert werden. Für ε-Transitionen wird keine Bewegungsrichtung notiert, da diese nur 0 sein kann. Eine weitere Ausnahme bilden eingehende Kanten, die die Formel Q_0 repräsentieren und somit die initialen Zustände markieren. Hier wird ebenfalls keine Bewegungsrichtung angegeben.

So kann zum Beispiel der Automat

$$
\begin{aligned}
&(\{a, b, c\}, && \textit{// Alphabet}\\
&\{q_0, q_1, q_2\}, && \textit{// Zustände}\\
&q_1 \wedge q_2, && \textit{// initiale Zustandsformel}\\
&\{((q_0, a), (q_1, -1) \wedge (q_2, 1) \vee \text{true}),\\
&\quad ((q_2, b), (q_0, 0)), ((q_0, c), \text{false}),\\
&\quad ((q_1, b), (q_2, 1))\}, && \textit{// Transitionen}\\
&\{(q_0, 0), (q_1, 0), (q_1, 1)\}) && \textit{// Paritäten}
\end{aligned}
$$

als Graph in Abbildung 3.6 auf der nächsten Seite dargestellt werden.

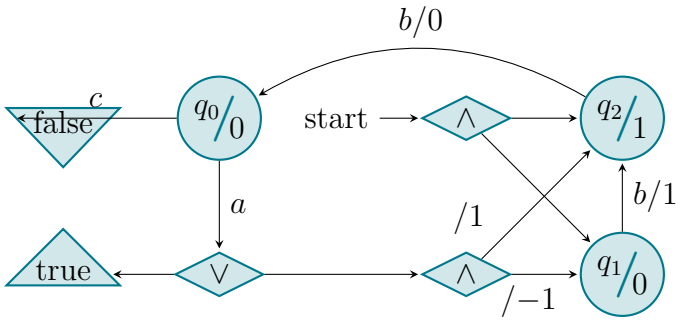

Abbildung 3.6.: Beispiel eines alternierenden Zwei-Wege-Paritätsautomaten in Graphendarstellung.

4. Umwandlung

Nachdem die benötigten Logiken und Automaten nun eingeführt sind, werden wir in diesem Kapitel die Umwandlung von regulärer Linearzeit-Temporallogik (RLTL) in einen alternierenden Zwei-Wege-Paritätsautomaten (2APW) betrachten. Dabei wird es primär um die Technik der Umwandlung und nicht um die konkrete Implementierung gehen. Diese wird im nächsten Kapitel erläutert.

Wir gehen im Folgenden immer davon aus, dass für die Umwandlung zusammen mit einem RLTL-Ausdruck auch ein Alphabet Σ gegeben ist. In der Praxis kann später auch aus dem RLTL-Ausdruck auf ein minimales Alphabet geschlossen werden.

4.1. Vom regulären Ausdruck zum 2NFA

Wie wir bereits gesehen haben, enthalten in RLTL die Power-Operatoren und die Sequenz-Operatoren reguläre Ausdrücke. Somit basieren nahezu alle RLTL-Ausdrücke auf regulären Ausdrücken. Um einen RLTL-Ausdruck in einen Automaten umwandeln zu können, müssen daher zunächst alle regulären Ausdrücke in Automaten umgewandelt werden können.

Für diese Umwandlung wird der unter anderem in [HMU01, S. 101–106] beschriebenen Algorithmus von Ken Thompson verwendet, den dieser 1968 als erster publizierte. Thompsons Algorithmus wurde für diese Arbeit auf die hier verwendete Definition von regulären Ausdrücken angepasst und mit der in [SL10, Lemma 3] gegebenen Umwandlung von regulären Ausdrücken mit Vergangenheit in Automaten kombiniert.

Thompsons Algorithmus basiert auf einem rekursiven Zusammensetzen von Teilausdrücken. Für jeden Operator wird ein Automat angegeben und für die Operanden können bereits bestimmte Automaten eingesetzt werden. Das Zusammensetzen dieser Teilautomaten erfolgt über ε-Transitionen, die Disjunktionen werden über nichtdeterministische Entscheidungen modelliert und reguläre Ausdrücke in die Vergangenheit werden über Zwei-Wege-Automaten realisiert. Somit entsteht bei dieser Umwandlung ein 2NFA mit ε-Transitionen. Um diesen Automaten in einen 2APW integrieren zu können, müssen vorher die ε-Transitionen entfernt werden, damit der 2NFA je nach Bedarf als 2NFA und als 2UFA verwendet werden kann.

Ein großer Vorteil dieses Verfahrens liegt in der einfachen Realisierbarkeit. Wird jeder Operator durch eine eigene Datenstruktur repräsentiert, kann jeder Operator eine Methode zur Verfügung stellen, um direkt durch den entsprechenden Automaten ersetzt zu werden. Da die Operanden direkt als Teilautomaten eingebaut werden können, kann die Umwandlung rekursiv sehr elegant implementiert werden.

Wir betrachten zunächst nur reguläre Ausdrücke ohne Vergangenheit, da nur solche klassischerweise vom Thompson-Algorithmus behandelt werden. Wir werden aber im Anschluss sehen, dass die Vergangenheit sehr elegant hinzugefügt werden kann.

In diesem Abschnitt wird schrittweise eine Funktion f angegeben, die von regulären Ausdrücken auf NFAs abbildet. Die Funktion ist wohldefiniert, da im Folgenden konkrete Ausgaben für alle möglichen Eingaben definiert werden.

4.1.1. Basisausdrücke

Bei den Basisausdrücken unterscheiden wir zwischen true, false und einem einzelnen Zeichen p.

$f(\text{true})$ liefert den Automaten

$$(\Sigma, \{q_0, q_1\}, \{q_0\}, \{((q_0, ?), q_1)\}, \{q_1\}).$$

Abbildung 4.1 zeigt $f(\text{true})$ als Graph.

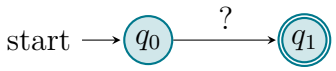

Abbildung 4.1.: $f(\text{true})$ als Graph.

$f(\text{false})$ liefert den Automaten

$$(\Sigma, \{q_0\}, \{q_0\}, \emptyset, \emptyset).$$

Abbildung 4.2 zeigt $f(\text{false})$ als Graph.

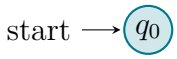

Abbildung 4.2.: $f(\text{false})$ als Graph.

$f(p)$ liefert den Automaten

$$(\Sigma, \{q_0, q_1\}, \{q_0\}, \{((q_0, p), q_1)\}, \{q_1\}).$$

Abbildung 4.3 auf Seite 44 zeigt $f(p)$ als Graph.

4.1.2. Disjunktion

Es seien x und y zwei reguläre Ausdrücke und $(\Sigma, Q_x, Q_{0,x}, \Delta_x, F_x) := f(x)$ sowie $(\Sigma, Q_y, Q_{0,y}, \Delta_y, F_y) := f(y)$ die zugehörigen Automaten. Dabei seien Q_x und Q_y vollständig disjunkt zueinander und zu den neuen Zuständen $\{q_s, q_f\}$. $f(x|y)$ liefert dann den Automaten

$$
\begin{aligned}
&(\Sigma,\\
&\{q_s, q_f\} \cup Q_x \cup Q_y,\\
&\{q_s\},\\
&\{((q_s, \varepsilon), q) \mid q \in Q_{0,x} \cup Q_{0,y}\} \cup \{((q, \varepsilon), q_e) \mid q \in F_x \cup F_y\} \cup \Delta_x \cup \Delta_y,\\
&\{q_f\}).
\end{aligned}
$$

Abbildung 4.4 auf Seite 44 zeigt $f(x|y)$ als schematischen Graphen.

4.1.3. Konkatenation

Es seien x und y zwei reguläre Ausdrücke und $(\Sigma, Q_x, Q_{0,x}, \Delta_x, F_x) := f(x)$ sowie $(\Sigma, Q_y, Q_{0,y}, \Delta_y, F_y) := f(y)$ die zugehörigen Automaten. Dabei seien Q_x und Q_y vollständig disjunkt zueinander. $f(xy)$ liefert dann den Automaten

$$
\begin{aligned}
&(\Sigma,\\
&Q_x \cup Q_y,\\
&Q_{0,x},\\
&\{((q, \varepsilon), q') \mid q \in F_x \wedge q' \in Q_{0,y}\} \cup \Delta_a \cup \Delta_b,\\
&F_y).
\end{aligned}
$$

Abbildung 4.5 auf Seite 44 zeigt $f(xy)$ als schematischen Graphen.

4.1.4. Binärer Kleene-Operator

Es seien x und y zwei reguläre Ausdrücke und $(\Sigma, Q_x, Q_{0,x}, \Delta_x, F_x) := f(x)$ sowie $(\Sigma, Q_y, Q_{0,y}, \Delta_y, F_y) := f(y)$ die zugehörigen Automaten. Dabei seien Q_x und Q_y vollständig disjunkt zueinander und zu den neuen Zuständen $\{q_s, q_f\}$. $f(x^*y)$ liefert

dann den Automaten

$$
\begin{aligned}
&(\Sigma,\\
&\{q_s, q_f\} \cup Q_x \cup Q_y,\\
&\{q_s\},\\
&\{((q_s, \varepsilon), q_f)\} \cup \{((q_s, \varepsilon), q) \mid q \in Q_{0,x}\}\\
&\qquad \cup \{((q, \varepsilon), q' \mid q \in F_x \wedge q' \in \{q_f\} \cup Q_{0,x}\}\\
&\qquad \cup \{((q_f, \varepsilon), q) \mid q \in Q_{0,y}\} \cup \Delta_x \cup \Delta_y,\\
&F_y).
\end{aligned}
$$

Abbildung 4.6 auf der nächsten Seite zeigt $f(x^*y)$ als schematischen Graphen.

4.1.5. Reguläre Ausdrücke mit Vergangenheit

Durch die bis hierhin spezifizierte Funktion f können alle regulären Ausdrücke ohne Vergangenheit in einen NFA umgewandelt werden. Um nun auch reguläre Ausdrücke mit Vergangenheit behandeln zu können, wird die Funktion f zu einer Funktion f_2 erweitert, die von regulären Ausdrücken mit Vergangenheit auf 2NFAs abbildet. Für die bisher betrachten Operatoren verhalte sich f_2 genau wie f mit dem Unterschied, dass alle Transitionen, die in f definiert wurden, in f_2 als Bewegung nach rechts interpretiert werden. Alle $((q, a), q') \in \Delta_f$ werden also zu $((q, a), (q', 1)) \in \Delta_{f_2}$.

Zusätzlich zu den aus f übernommenen Operatoren, ist f_2 auch auf $-p$ für einen Basisausdruck p definiert. Für die Umwandlung von $-p$ in einen 2NFA wird hier das Verfahren aus [SL10, Lemma 3] verwendet.

$f_2(-p)$ liefert den Automaten

$$(\Sigma, \{q_0, q_1, q_2\}, \{q_0\}, \{((q_0, ?), (q_1, -1)), ((q_1, p), (q_2, 0))\}, \{q_2\}).$$

Abbildung 4.7 auf der nächsten Seite zeigt $f_2(-p)$ als schematischen Graphen.

An dieser Stelle sei noch einmal daran erinnert, dass der Vergangenheits-Operator $(\cdot)^{-1}$ auf beliebigen regulären Ausdrücken in den Vergangenheits-Operator $-(\cdot)$ auf Basisausdrücken umgewandelt werden kann. Daher ist die hier angegebene Umwandlung ausreichend, um alle regulären Ausdrücke mit Vergangenheit in einen 2NFA umzuwandeln.

Anders als bei den bisher erzeugten NFAs ist die Semantik der einzelnen Bausteine aus denen die 2NFAs zusammengesetzt werden nicht immer sinnvoll. Insbesondere ist der Automat aus Abbildung 4.7 auf der nächsten Seite ohne Kontext nicht sinnvoll, da die einzig mögliche erste Transition eine Bewegung nach links enthält. Ein

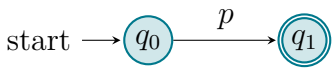

Abbildung 4.3.: $f(p)$ als Graph.

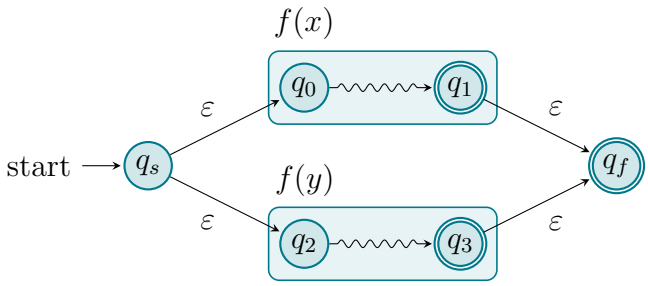

Abbildung 4.4.: $f(x|y)$ als schematischer Graph.

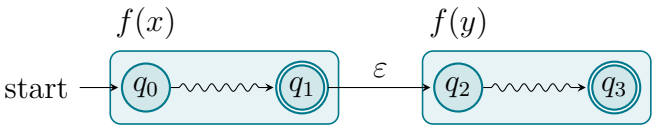

Abbildung 4.5.: $f(xy)$ als schematischer Graph.

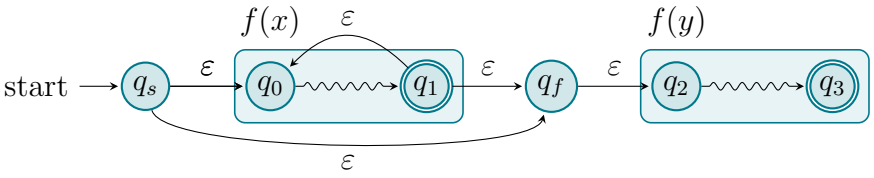

Abbildung 4.6.: $f(x^*y)$ als schematischer Graph.

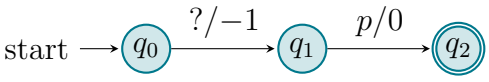

Abbildung 4.7.: $f_2(-p)$ als Graph.

Schritt nach links in der initialen Konfiguration $(q_0, (w, i, j))$ liefert eine Konfiguration $(q_1, (w, i-1, j))$, die in einem akzeptierenden Lauf verboten ist. Anschaulich wird durch diesen Schritt das Wort nach links verlassen, sodass es sofort verworfen wird. Somit akzeptiert dieser Automat genau die leere Sprache. Wird dieser Automat aber als Baustein in weiteren Automaten verwendet, so ist q_0 nicht mehr Teil der initialen Konfiguration, sodass durchaus sinnvolle Automaten entstehen können.

4.1.6. Entfernung der ε-Transitionen

Beim Einbau in den 2APW wird der durch die hier vorgestellte Umwandlungsfunktion f_2 erzeugte 2NFA in manchen Situationen als universeller Automat, also als 2UFA, interpretiert. Insbesondere bei der Erzeugung des negierten Automaten kann auf diese Weise für alle möglichen Pfade durch den NFA ein Erfolg ausgeschlossen werden. Ein 2NFA kann sehr leicht als 2UFA interpretiert werden, indem die möglichen Folgezustände $\delta(q, a)$ für einen Zustand q und ein gelesenes Zeichen a nicht als Veroderung

$$\bigvee_{q' \in \delta(q,a)} q'$$

sondern als Verundung

$$\bigwedge_{q' \in \delta(q,a)} q'$$

in die benötigte positive boolesche Formel umgewandelt werden. (Für einen Zwei-Wege-Automaten gilt dabei eigentlich $q' \in \hat{Q}$ mit $\hat{Q} = Q \times \{-1, 0, 1\}$.) Dadurch entsteht ein 2APW, bei dem nur ein möglicher Lauf mit vielen möglichen Pfaden existiert. Da in einem akzeptierenden Lauf alle Pfade akzeptierend sein müssen, wird der 2NFA durch diese Behandlung zum universellen Automaten. Entsprechend der Definition von 2APWs mit ε-Transitionen bildet eine ε-Transition aber immer einen eigenen alternativen Lauf und keinen neuen Pfad im bestehenden Lauf. Damit also auch die Alternativen, die durch die ε-Transitionen entstehen, bei der Verwendung des 2NFAs als universellen Automaten als zusätzlicher Pfad im Lauf des APWs und nicht als eigener Lauf genutzt werden, müssen die ε-Transitionen im 2NFA entfernt werden, bevor der 2NFA in den 2APW eingebaut wird.

In der Automatentheorie wird häufig nur die Mächtigkeit von Automaten betrachtet, um die Äquivalenz von deterministischen Automaten zu Automaten mit ε-Transitionen, nichtdeterministischen Automaten und Zwei-Wege-Automaten zu betrachten. Vor diesem Hintergrund ist es kein besonders interessantes Problem, einen

2NFA mit ε-Transitionen in einen 2NFA umzuwandeln, da die ε-Transitionen einfach bei der Umwandlung in einen deterministischen endlichen Automaten mit behandelt werden können. Die damit einhergehende drastische Erweiterung der Zustandsmenge ist aber für diese Arbeit nicht akzeptabel, da nichtdeterministische Transitionen im 2APW nativ ausgedrückt werden können, sodass eine Umwandlung in einen deterministischen endlichen Automaten nicht benötigt wird. Eine weitere Möglichkeit, ε-Transitionen zu entfernen, besteht in der Darstellung als Transition im Zwei-Wege-Automaten mit Bewegungsrichtung 0 für ein beliebiges Eingabezeichen. Durch diese Variante würde allerdings für nahezu alle regulären Ausdrücke ein 2NFA entstehen, während bisher nur für reguläre Ausdrücke mit Vergangenheit ein 2NFA generiert wird. Die weitere Verarbeitung eines APWs ist aber deutlich einfacher als die eines 2APWs, sodass das Ausweichen auf Zwei-Wege-Automaten keine gute Option ist. Daher wurde im Rahmen dieser Arbeit ein Verfahren entwickelt, wie die ε-Transitionen im (2)NFA unter Verwendung des Nichtdeterminismus mit einer minimalen Erweiterung der Zustandsmenge entfernt werden können.

Die prinzipielle Idee dieses Verfahrens besteht darin, die ε-Transition vorweg zu nehmen. Eine Transition, die in einem Zustand endet, von dem eine ε-Transition ausgeht, kann auch gleich im Zielzustand dieser ε-Transition enden. Je nach Art der (anderen) ausgehenden Transitionen kann auf diese Weise auf den Zustand, in dem die ε-Transition beginnt, sogar ganz verzichtet werden. Weitere Überlegungen werden dann nötig, wenn der Automat mehrere ε-Transitionen direkt hintereinander aufweist, oder wenn durch dieses Vorwegnehmen von ε-Transitionen und das damit einhergehende Auslassen von Zuständen akzeptierende Zustände ausgelassen werden. Wir wollen daher im Folgenden eine formale Beschreibung des Verfahrens betrachten. Dazu sei ein NFA $(\Sigma, Q, Q_0, \Delta, F)$ gegeben.

Im ersten Schritt wird eine partielle Funktion $t : Q \rightsquigarrow 2^{Q \times \mathcal{B}^+(\emptyset)}$ erzeugt, die für jede ε-Transition vom Startzustand auf die Menge aller Zielzustände abbildet. Diese Zielzustände sind dabei jeweils mit der Akzeptanzbedingung für den Startzustand annotiert. Statt $q \in Q$ wird also das Paar $(q, q \in F)$ gespeichert. Die zusätzliche Speicherung der Akzeptanzbedingung ist wichtig, damit für jeden Zielzustand einzeln vermerkt werden kann, welche Akzeptanzbedingung dieser haben muss, um alle durch das Vorwegnehmen der ε-Transition übersprungenen Zustände ersetzen zu können.

Im zweiten Schritt wird eine Art Hülle dieser Ersetzungsfunktion t gebildet. Dazu sei t_1 die Funktion t, bei der in allen Funktionswerten von t_1 Zustände q, für die t definiert ist, durch $t(q)$ ersetzt wurden. Entsprechend sei t_{i+1} die Funktion t_i, bei der in allen Funktionswerten von t_i Zustände q, für die t_i definiert ist, durch $t_i(q)$ ersetzt wurden. Diese Iteration wird fortgesetzt, bis ein t_k entsteht, bei dem alle Funktionswerte von t_k nur noch Zustände q enthalten, für die t_k nicht definiert ist. Ein solches t_k existiert immer, da wir ohne Beschränkung der Allgemeinheit

vorraussetzen können, dass der Automat frei von ε-Zyklen ist. Beim Ersetzen eines Paares (q, a) bestehend aus einem Zustand q und einer Akzeptanzbedingung a in einem Funktionswert von t_i durch $t_i(q)$ werden in $t_i(q)$ alle Akzeptanzbedingungen durch die Disjunktion aus der alten und der neuen Akzeptanzbedingung ersetzt:

$$\begin{aligned} t_{i+1}(q) :=& \{(q'', a'' \vee a') \mid (q', a') \in t_i(q) \wedge (q'', a'') \in t_i(q')\} \\ & \cup \{(q', a') \mid (q', a') \in t_i(q) \wedge t_i(q') \text{ nicht definiert}\}. \end{aligned}$$

Im dritten Schritt werden nun Paare von Zuständen q und Akzeptanzbedingungen a in den Funktionswerten von t_k gesucht, die eine höhere Akzeptanzbedingung a fordern, als der Zustand q tatsächlich hat. Konkret werden also Paare (q, true) gesucht mit $q \notin F$. Solche Fälle müssen gelöst werden, da sonst die Akzeptanzbedingungen der Läufe verändert würden. Für die Lösung dieses Problems unterscheiden wir zwei Fälle in Abhängigkeit des betroffenen Zustandes q:

1. Der Zustand q hat nur eine eingehende Transition. Diese Transition muss in diesem Fall die ε-Transition sein, die entfernt wird. Vor der Entfernung enthielten also alle Läufe, die diese Transition enthielten, auch diese eine ε-Transition. In diesem Fall ist die Lösung einfach: Der Zustand q wird der Menge der akzeptierenden Zustände hinzugefügt.

2. Der Zustand q hat mehrere eingehende Transitionen. In diesem Fall muss ein zusätzlicher Zustand q' erzeugt werden. Dieser neue Zustand wird der Menge der akzeptierenden Zustände hinzugefügt und für alle ausgehenden Transitionen des Zustandes q wird eine entsprechende ausgehende Transition für den Zustand q' der Transitionsrelation Δ hinzugefügt. Jetzt wird die Funktion t_k so angepasst, dass das problematische Paar (q, true) durch (q', true) ersetzt wird. Der neue Zustand q' entspricht also gerade dem Zustand q mit dem Unterschied, dass q' nun die Forderungen an die Akzeptanzbedingung erfüllt.

Im vierten Schritt wenden wir die Ersetzungsfunktion t_k nun auf die Transitionsrelation Δ des im letzten Schritt angepassten NFA an. Dabei werden alle ε-Transitionen entfernt und für alle Transitionen zu einem Zustand q, für den $t_k(q)$ definiert ist, werden Transitionen zu allen Zuständen in $t_k(q)$ hinzugefügt. Die ursprüngliche Transition zu q wird nur dann aufgehoben, wenn der Zustand q bleibende ausgehende Transitionen, also keine ε-Transitionen, besitzt.

Im fünften Schritt wenden wir die Ersetzungsfunktion t_k schließlich auch auf Q_0 an. Dabei werden auch hier für alle Zustände q, für die $t_k(q)$ definiert ist, die Zustände in $t_k(q)$ ergänzt und q nur dann aufgehoben, wenn q bleibende ausgehende Transitionen hat. Im vierten und fünften Schritt können die Akzeptanzbedingungen, die jeweils mit den Zuständen gespeichert sind, ignoriert werden, denn der vierte Schritt hat bereits sichergestellt, dass diese eingehalten werden.

Beim Anwenden der Ersetzungsfunktion im vierten und fünften Schritt können unerreichbare Zustände entstehen. Konnte ein Zustand vor der Entfernung der ε-Transitionen nur durch eine solche erreicht werden, so kann dieser Zustand nun nicht mehr erreicht werden, da alle ε-Transitionen vorweg genommen wurden. Diese Zustandsreste können im sechsten Schritt entfernt werden, indem von der initialen Zustandsformel Q_0 aus alle erreichbaren Zustände mit einer Tiefensuche markiert werden. Alle Zustände die nun nicht markiert wurden, können entfernt werden. Ebenso alle Transitionen, die von einem nicht markierten Zustand ausgehen. Diese Tiefensuche ist deutlich effizienter als Algorithmen, die immer wieder über alle Zustände iterieren und alle Zustände ohne eingehende Transitionen entfernen, denn bei diesem Vorgehen müsste die Iteration so lange wiederholt werden, bis in einer ganzen Iteration kein Zustand mehr entfernt wurde.

Das ganze Verfahren kann genauso auf 2NFAs angewendet werden. Die Ersetzungsfunktion t bildet noch immer nur auf Mengen von Paaren von Zuständen und Akzeptanzbedingungen ab, da ε-Transitionen nur die Bewegungsrichtung 0 haben können. Die einzige Anpassung des Verfahrens muss im vierten Schritt bei der Anwendung der Ersetzungsfunktion t_k auf die Transitionsrelation Δ erfolgen. Hier muss beim 2NFA die Bewegungsrichtung der ursprünglichen Transition erhalten werden.

Beispiel

Als Beispiel diene der NFA in Abbildung 4.8. Dieser Automat ist ein interessantes Beispiel, da der Zustand a_1 akzeptierend ist und beim Entfernen der ε-Transitionen übersprungen wird. Das neue Ziel b_1 ist aber nicht akzeptierend und kann es auch nicht werden, da eine Transition von b_2 zu b_1 existiert.

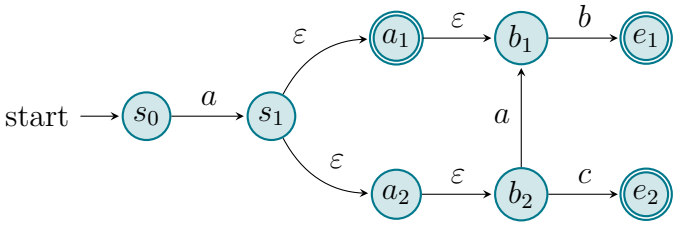

Abbildung 4.8.: Ausgangsautomat, an dem die Entfernung von ε-Transitionen demonstriert wird. Dieser NFA akzeptiert die Worte a, ab, ac und aab.

Wir beginnen im ersten Schritt damit, die Funktion t aufzustellen. Tabelle 4.1 auf Seite 50 repräsentiere alle Elemente aus t.

Da $t(a_1)$ und $t(a_2)$ definiert sind, werden die Einträge (a_1, false) und (a_2, false) im zweiten Schritt entsprechend ersetzt. Da $t(a_1) = \{(b_1, \text{true})\}$ gilt, wird (a_1, false)

durch $(b_1, \text{false} \vee \text{true}) = (b_1, \text{true})$ ersetzt. Wir erhalten die Funktion t_1, die durch Tabelle 4.2 auf der nächsten Seite repräsentiert sei.

Da diese Funktion (b_1, true) enthält und der Zustand b_1 nicht akzeptierend werden kann, wird im dritten Schritt ein akzeptierender Zustand b_1' erzeugt. Abbildung 4.9 zeigt das Ergebnis und Tabelle 4.3 auf der nächsten Seite zeigt die Funktion t_1 nach der Anpassung auf den neuen Zustand b_1'.

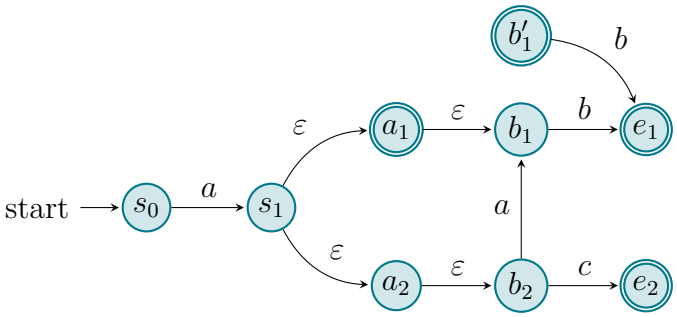

Abbildung 4.9.: Der Beispielautomat nach der Erzeugung des zusätzlichen Zustandes b_1'. Dieser Zustand hat alle ausgehenden Transitionen, die b_1 auch hat, ist aber zusätzlich akzeptierend. Da dieser Zustand noch nicht erreichbar ist, wird durch seine Ergänzung die Menge der akzeptierten Worte nicht verändert.

In Abbildung 4.10 auf der nächsten Seite wurden nun im vierten Schritt alle ε-Transitionen entfernt und in allen Transitionen die Zustände, für die t_1 definiert ist, durch die entsprechenden Transitionen ersetzt. Im fünften Schritt wird der Automat nicht weiter angepasst, da Q_0 keine Zustände enthält, für die t_1 definiert ist.

In Abbildung 4.11 auf Seite 53 wurden schließlich im sechsten Schritt die nun nicht mehr erreichbaren Zustände entfernt. Dieser NFA akzeptiert die gleichen Worte wie der Ausgangsautomat, enthält aber keine ε-Transitionen mehr und die Menge der Zustände wurde dabei sogar reduziert.

4.2. Vom RLTL-Ausdruck zum 2APW

Mit der Funktion f_2 sind wir nun in der Lage einen regulären Ausdruck mit Vergangenheit in einen 2NFA umzuwandeln. Wir können im Folgenden also davon ausgehen, dass die in RLTL-Ausdrücken enthaltenen regulären Ausdrücke bereits als 2NFA vorliegen und in der weiteren Konstruktion verwendet werden können.

Für die Umwandlung von RLTL-Ausdrücken in Automaten geben wir analog zu f eine Funktion g an, die von RLTL-Ausdrücken auf 2APWs abbildet. Das hier

q	$t(q)$
s_1	$\{(a_1, \text{false}), (a_2, \text{false})\}$
a_1	$\{(b_1, \text{true})\}$
a_2	$\{(b_2, \text{false})\}$

Tabelle 4.1.: Repräsentation der initialen Funktion t vor der ersten Hüllenbildung.

q	$t(q)$
s_1	$\{(b_1, \text{true}), (b_2, \text{false})\}$
a_1	$\{(b_1, \text{true})\}$
a_2	$\{(b_2, \text{false})\}$

Tabelle 4.2.: Repräsentation der Funktion t_1, die durch die Hüllenbildung entsteht.

q	$t(q)$
s_1	$\{(b'_1, \text{true}), (b_2, \text{false})\}$
a_1	$\{(b'_1, \text{true})\}$
a_2	$\{(b_2, \text{false})\}$

Tabelle 4.3.: Repräsentation der Funktion t_1 nach der Anwendung des dritten Schrittes zur Anpassung der Zustände an die geforderten Akzeptanzbedingungen.

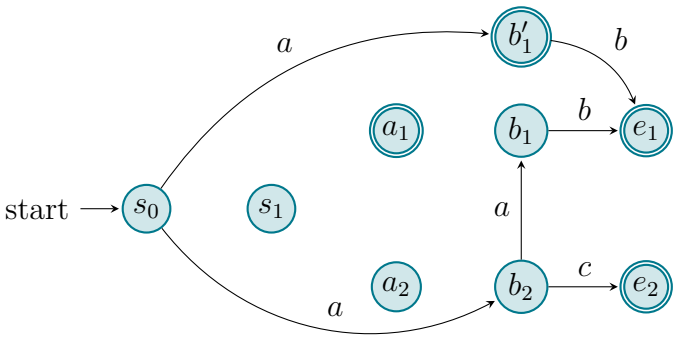

Abbildung 4.10.: Der Beispielautomat nach der Entfernung aller ε-Transitionen und der Anwendung der Ersetzungsfunktion t_1. Durch diese Operationen sind die Zustände s_1, a_1 und a_2 nun nicht mehr erreichbar und werden im nächsten Schritt entfernt. Der Automat akzeptiert noch immer genau die Worte a, ab, ac und aab.

verwendete Verfahren wurde von César Sánchez und Julián Samborski-Forlese entwickelt und in [SSF11] vorgestellt. Ganz ähnlich zur Umwandlung von regulären Ausdrücken, werden wir wieder ein Bottom-Up-Verfahren angeben, dass für jeden Operator einen Automaten angibt, der aus Teilautomaten besteht, die den Operanden entsprechen. Diese Teilautomaten entsprechen direkt den umgewandelten RLTL-Ausdrücken der Operanden und müssen nicht weiter angepasst werden. Dieses Verfahren lässt sich sehr elegant implementieren, da jeder Operator unabhängig von anderen Operatoren der Formel transformiert werden kann.

Die Umwandlung regulärer Ausdrücke in Automaten ist unter anderem auch deswegen so elegant möglich, da keine Negationen in regulären Ausdrücken vorkommen. Daher können Teilautomaten im Verfahren immer ohne Anpassung direkt weiter verwendet werden. Bei einer Negation müsste der Teilautomat hingegen negiert werden. Im Gegensatz zu regulären Ausdrücken enthalten RLTL-Ausdrücke Negationen. Um das Problem, einen Teilautomaten negieren zu müssen, zu umgehen, verwendet die Umwandlung folgende Methode: Wir wandeln RLTL-Ausdrücke nicht in einen 2APW um, sondern in ein Paar aus zwei gespiegelten 2APWs (A,B). Dabei gilt $\mathcal{L}(A) = \overline{\mathcal{L}(B)}$. Die beiden Automaten akzeptieren gerade jeweils die komplementäre Sprache des anderen Automaten. Die Funktion g bildet also tatsächlich nicht auf 2APWs, sondern auf 2-Tupel von 2APWs ab. Die Negation in einem Ausdruck $\neg x$ für einen beliebigen RLTL-Ausdruck x kann nun einfach umgewandelt werden durch vertauschen der beiden Automaten im Tupel:

$$f(\neg p) = (B, A) \quad \text{für} \quad f(p) = (A, B).$$

Formal sei also $g : \texttt{RLTL} \to \texttt{2APW} \times \texttt{2APW}$, wobei RLTL die Menge aller Ausdrücke der regulären linearen Temporallogik und 2APW alle endlichen alternierenden Zwei-Wege-Paritätsautomaten beschreibt. Aus technischen Gründen – insbesondere zur Verwendung in den folgenden Abbildungen – definieren wir für $g(a) = (A, \overline{A})$ die Funktionen $g_\top(a) := A$ und $g_\bot(a) = \overline{A}$, die jeweils auf den ersten bzw. zweiten Automaten aus $g(a)$ abbilden.

g bildet zwar auf Zwei-Wege-Automaten ab, die folgende Umwandlung wird aber selbst nur Schritte nach rechts definieren. Andere Bewegungsrichtungen kommen nur durch die Verwendung von 2NFAs hinzu. Als regulärer Ausdruck für den Verzögerungsoperanden in Power-Operatoren können aber insbesondere auch Konstruktionen wie $-p$ vorkommen, die als allein stehender Automat nur die leere Sprache akzeptieren, aber im Kontext eines größeren Automaten durchaus komplexe Strukturen ermöglichen. Daher ist es nicht möglich, die verwendeten 2NFAs vor der Verwendung im 2APW in einen NFA umzuwandeln. Somit wird der volle Umfang von 2APWs durchaus benötigt, auch wenn in der folgenden Umwandlung nur Bewegungsschritte nach rechts vorkommen werden.

Eine weitere Besonderheit dieses Verfahrens ist die Einschränkung, dass 2APWs mit nur drei Farben entstehen. Wie bereits dargelegt, kann diese spezielle Form von alternierenden Paritätsautomaten auch als Streett-Automat mit nur einem Tupel verstanden werden.

Im Folgenden geben wir induktiv für die Funktion g konkrete Ausgaben für alle möglichen Eingaben an.

4.2.1. Leere Sprache

$g(\emptyset)$ liefert das gespiegelte 2APW-Paar

$$((\Sigma, \{q_0\}, q_0, \{((q_0, ?), \text{false}), \{(q_0, 0)\}\}),$$
$$(\Sigma, \{q_0\}, q_0, \{((q_0, ?), \text{true}), \{(q_0, 0)\}\})).$$

Abbildung 4.12 auf der nächsten Seite zeigt $g(\emptyset)$ als Paar von Graphen. Intuitiv akzeptiert der erste Automat des Tupels keine Eingabe und der zweite Automat alle Eingaben, sodass diese gerade der leeren Sprache bzw. der negierten leeren Sprache entsprechen. Diese Automaten sind insbesondere als Hebel in die Unendlichkeit für NFAs sehr wichtig. Reguläre Ausdrücke können mit dem Konkatenations-Operator in einen RLTL-Ausdruck verwandelt werden, wenn sie mit $\neg\emptyset$ kombiniert werden.

4.2.2. Disjunktion und Konjunktion

Es seien a und b zwei RLTL-Ausdrücke und

$$((\Sigma, Q_a, Q_{0,a}, \delta_a, F_a), (\Sigma, \overline{Q_a}, \overline{Q_{0,a}}, \overline{\delta_a}, \overline{F_a})) := g(a)$$

sowie

$$((\Sigma, Q_b, Q_{0,b}, \delta_b, F_b), (\Sigma, \overline{Q_b}, \overline{Q_{0,b}}, \overline{\delta_b}, \overline{F_b})) := g(b)$$

die zugehörigen gespiegelten Automatenpaare. Dabei seien Q_a und Q_b vollständig disjunkt zueinander. Bei der Disjunktion liefert $g(a \vee b)$ dann das gespiegelte 2APW-Paar

$$((\Sigma, Q, Q_0, \delta, F), (\Sigma, \overline{Q}, \overline{Q_0}, \overline{\delta}, \overline{F}))$$

mit

$$\begin{aligned} Q &= Q_a \cup Q_b & \qquad \overline{Q} &= \overline{Q_a} \cup \overline{Q_b} \\ Q_0 &= Q_{0,a} \vee Q_{0,b} & \qquad \overline{Q_0} &= \overline{Q_{0,a}} \wedge \overline{Q_{0,b}} \\ \delta &= \delta_a \cup \delta_b & \qquad \overline{\delta} &= \overline{\delta_a} \cup \overline{\delta_b} \\ F &= F_a \cup F_b & \qquad \overline{F} &= \overline{F_a} \cup \overline{F_b} \end{aligned}$$

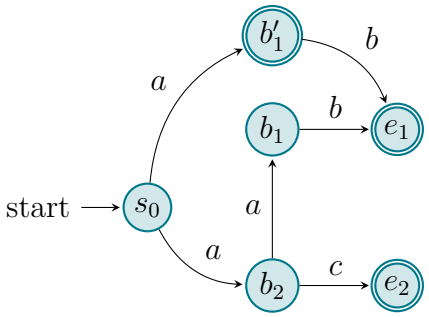

Abbildung 4.11.: Das Ergebnis der Entfernung der ε-Transitionen. Die Menge der akzeptierten Worte wurde auch durch den letzten Schritt nicht mehr verändert, da nur nicht erreichbare Zustände entfernt wurden.

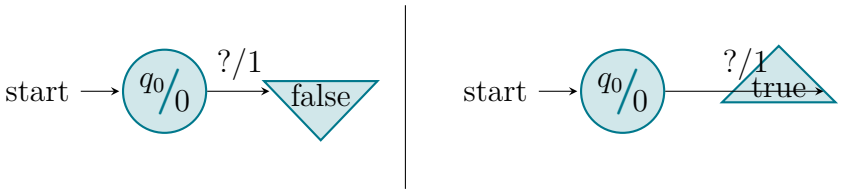

Abbildung 4.12.: $g(\emptyset)$ als Paar von Graphen.

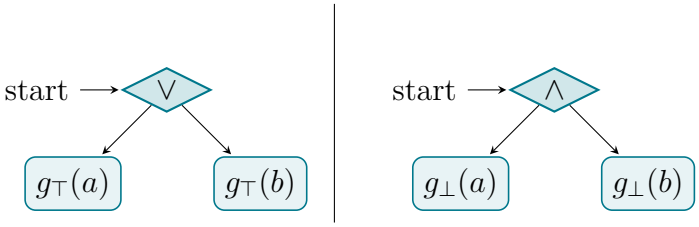

Abbildung 4.13.: $g(a \vee b)$ als Paar von schematischen Graphen.

Abbildung 4.13 auf der vorherigen Seite zeigt $g(a \vee b)$ als schematischen Graphen.

Die Konjunktion $g(a \wedge b)$ kann bis auf Vertauschen von $\wedge$ und $\vee$ genau wie die Disjunktion behandelt werden. Es ändert sich dabei nur

$$Q_0 = Q_{0,a} \wedge Q_{0,b} \qquad \overline{Q_0} = \overline{Q_{0,a}} \vee \overline{Q_{0,b}}$$

Abbildung 4.14 zeigt $g(a \wedge b)$ als schematischen Graphen.

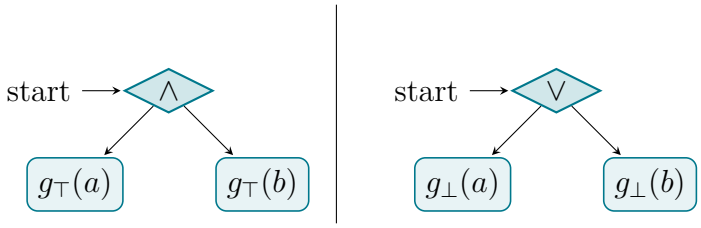

Abbildung 4.14.: $g(a \wedge b)$ als Paar von schematischen Graphen.

4.2.3. Konkatenation

Es seien x ein regulärer Ausdruck, a ein RLTL-Ausdruck und

$$(\Sigma, Q_x, Q_{0,x}, \Delta_x, F_x) := f_2(x)$$

der zugehörige 2NFA, sowie

$$((\Sigma, Q_a, Q_{0,a}, \delta_a, F_a), (\Sigma, \overline{Q_a}, \overline{Q_{0,a}}, \overline{\delta_a}, \overline{F_a})) := g(a)$$

das zugehörige gespiegelte Automatenpaar. Dabei seien Q_x und Q_a vollständig disjunkt zueinander und der 2NFA liege als totaler Automat vor. $g(x; a)$ liefert dann das gespiegelte 2APW-Paar

$$((\Sigma, Q, Q_0, \delta, F), (\Sigma, \overline{Q}, \overline{Q_0}, \overline{\delta}, \overline{F}))$$

mit

$$\begin{aligned} Q &= Q_x \cup Q_a \\ Q_0 &= \text{join}(\vee, Q_{0,x}) \\ \delta(q,s) &= \begin{cases} \text{inject}(\vee, \delta_x(q,s), F_x, Q_{0,a}) & \text{wenn } q \in Q_x \\ \delta_a(q,s) & \text{wenn } q \in Q_a \end{cases} \\ F &= \{(q,1) \mid q \in Q_x\} \cup F_a \end{aligned}$$

und

$$\begin{aligned}
\overline{Q} &= Q_x \cup \overline{Q_a} \\
\overline{Q_0} &= \text{join}(\wedge, \overline{Q_{0,x}}) \\
\overline{\delta}(q,s) &= \begin{cases} \text{inject}(\wedge, \delta_x(q,s), F_x, \overline{Q_{0,a}}) & \text{wenn } q \in Q_x \\ \overline{\delta_a}(q,s) & \text{wenn } q \in \overline{Q_a} \end{cases} \\
\overline{F} &= \{(q,0) \mid q \in Q_x\} \cup \overline{F_a}.
\end{aligned}$$

Dabei seien die Funktionen join : $\{\vee, \wedge\} \times 2^{\mathcal{B}^+(Q)} \to \mathcal{B}^+(Q)$ und join : $\{\vee, \wedge\} \times \mathcal{B}^+(\hat{Q}) \to 2^{\mathcal{B}^+(\hat{Q})}$ mit $\hat{Q} = Q \times \{-1, 0, 1\}$ gegeben durch

$$\begin{aligned}
\text{join}(\vee, R) &= \bigvee_{r \in R} r \\
\text{join}(\wedge, R) &= \bigwedge_{r \in R} r.
\end{aligned}$$

Die Funktion join erzeugt also einen positiven booleschen Ausdruck aus einer Menge von Ausdrücken oder Zuständen, indem alle Elemente aus der gegebenen Menge durch den gegebenen Operator verbunden werden.

Weiter sei die Funktion

$$\text{inject} : \{\vee, \wedge\} \times 2^{\hat{Q}} \times 2^{Q} \times \mathcal{B}^+(Q) \to \mathcal{B}^+(\hat{Q}) \quad \text{mit } \hat{Q} = Q \times \{-1, 0, 1\}$$

gegeben durch

$$\begin{aligned}
\text{inject}(u, R, F, Q) = \text{join}(u, &\{(f,d)u(q,d) \mid (f,d) \in R \wedge f \in F \wedge q \in Q\} \\
&\cup \{(q,d) \mid (q,d) \in R \wedge q \notin F\}).
\end{aligned}$$

Für eine Operation u, eine Menge von mit Leserichtungen annotierten Zuständen R, eine Menge von besonderen Zuständen F und einer weiteren Menge von Zuständen Q liefert die Funktion inject(u, R, F, Q) unter Verwendung von join die positive boolesche Formel über mit Leserichtungen annotierten Zuständen, die alle Zustände aus R enthält. Zusätzlich werden für alle Zustände aus R, die auch in F vorkommen und auf diese Weise als besondere Zustände markiert sind, die Zustände aus Q mit eingebaut. Dabei wird die annotierte Bewegungsrichtung der Zustände aus R für die zusätzlichen Zustände aus Q übernommen. Auf diese Weisen können alle Transitionen zu speziellen Zuständen um eine zusätzliche Transition zu allen Zuständen aus Q erweitert werden. Diese Funktion wird hier benutzt, um alle Transitionen zu akzeptierenden Zuständen im 2NFA um eine Transition zu allen Startzuständen des 2APW zu ergänzen. Dabei wird die Bewegungsrichtung der Transition aus dem 2NFA erhalten.

Abbildung 4.15 auf Seite 57 zeigt $g(x; a)$ als schematischen Graphen.

Für das zweite Element des Paares, also den negierten Automaten, wird dabei der 2NFA als 2UFA interpretiert, indem mehrere Transitionen bei gleichem Zustand und Eingabezeichen zu einer großen Konjunktion in der durch join erzeugten positiven booleschen Formel werden. Auf diese Weise kann für alle Fälle, in denen der 2NFA die Möglichkeit hatte, das Wort zu akzeptieren, eine Bedingung überprüft werden. Wird kein solcher Fall erreicht, wird unendlich lange eine Schleife innerhalb des 2NFA durchlaufen. Da die Zustände des 2NFA die Praität 0 haben, führt auch das unendlich lange Verbleiben im 2NFA zu einem akzeptierenden Pfad. Damit dieses Verfahren funktioniert, ist es wichtig, dass der 2NFA als totaler Automat vorliegt. Sonst wäre ein solches unendlich langes Verbleiben ohne einen der akzeptierenden Zustände zu erreichen oft nicht möglich.

4.2.4. Duale Konkatenation

Die duale Konkatenation $g(x;;a)$ kann bis auf Vertauschung der Paritäten für den 2NFA und Vertauschung von $\wedge$ und $\vee$ genau wie die Konkatenation behandelt werden. Es werden hier nur die Änderungen gegenüber der Konkatenation angegeben.

$$
\begin{aligned}
Q_0 &= \text{join}(\wedge, Q_{0,x}) \\
\delta(q,s) &= \begin{cases} \text{inject}(\wedge, \delta_x(q,s), F_x, Q_{0,a}) & \text{wenn } q \in Q_x \\ \delta_a(q,s) & \text{wenn } q \in Q_a \end{cases} \\
F &= \{(q,0) \mid q \in Q_x\} \cup F_a
\end{aligned}
$$

und

$$
\begin{aligned}
\overline{Q_0} &= \text{join}(\vee, \overline{Q_{0,x}}) \\
\overline{\delta}(q,s) &= \begin{cases} \text{inject}(\vee, \delta_x(q,s), F_x, \overline{Q_{0,a}}) & \text{wenn } q \in Q_x \\ \overline{\delta_a}(q,s) & \text{wenn } q \in \overline{Q_a} \end{cases} \\
\overline{F} &= \{(q,1) \mid q \in Q_x\} \cup \overline{F_a}
\end{aligned}
$$

Abbildung 4.16 auf der nächsten Seite zeigt $g(x;;a)$ als schematischen Graphen.

Man erkennt sofort, dass die duale Konkatenation sehr analog zur Konkatenation behandelt werden kann. Es werden Konjunktion, Disjunktion und die Paritäten vertauscht. Diese einfach Behandlung in der Umwandlung durch die geschickte Konstruktion der dualen Operatoren werden wir bei den Power-Operatoren wiederfinden.

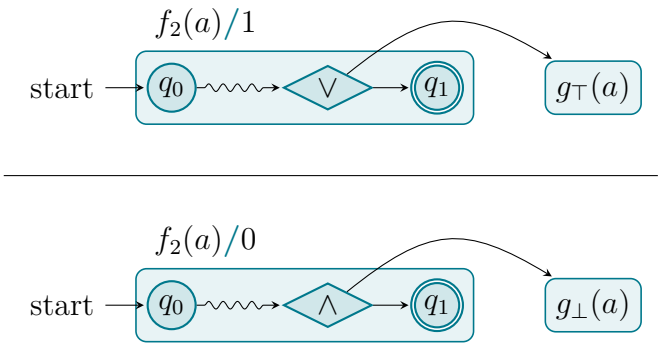

Abbildung 4.15.: $g(x; a)$ als Paar von schematischen Graphen.

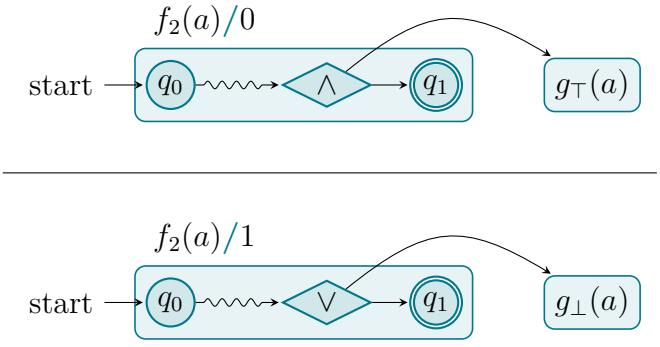

Abbildung 4.16.: $g(x; ; a)$ als Paar von schematischen Graphen.

4.2.5. Power-Operatoren

Wir betrachten nun die Transformationsfunktion g für die verschiedenen Power-Operatoren. Da für den Power-Operator, den schwachen Power-Operator und die beiden dazu dualen Power-Operatoren jeweils ein gespiegeltes APW-Paar entsteht, müssten insgesamt acht Automaten beschrieben werden, die in ihrer grundsätzlichen Struktur alle sehr ähnlich konstruiert werden. Um diese Redundanz zu vermeiden, geben wir im Folgenden einen parametrisierten Automaten für alle acht Fälle an.

Es seien x ein regulärer Ausdruck und a und b RLTL-Ausdrücke und

$$(\Sigma, Q_x, Q_{0,x}, \Delta_x, F_x) := f_2(x)$$

der zugehörige 2NFA, sowie

$$((\Sigma, Q_a, Q_{0,a}, \delta_a, F_a), (\Sigma, \overline{Q_a}, \overline{Q_{0,a}}, \overline{\delta_a}, \overline{F_a})) := g(a)$$

und

$$((\Sigma, Q_b, Q_{0,b}, \delta_b, F_b), (\Sigma, \overline{Q_b}, \overline{Q_{0,b}}, \overline{\delta_b}, \overline{F_b})) := g(b)$$

die zugehörigen gespiegelten Automatenpaare. Dabei seien Q_x, Q_a und Q_b vollständig disjunkt zueinander und zu dem neuen Zustand $\{q_0\}$.

$g(a/x\rangle\rangle b)$, $g(a/x\rangle b)$, $g(a//x\rangle b)$ und $g(a//x\rangle\rangle b)$ liefern nun gespiegelte 2APW-Paare. Die Automaten sind alle von der gleichen Struktur. Wir geben daher einen Automaten in Abhängigkeit der folgenden Parameter an.

- Der Parameter $A = (\Sigma, Q_a, Q_{0,a}, \delta_a, F_a)$ sei ein 2APW, der aus der Teilformel a entstanden ist. Es gelte entweder $A = g_\top(a)$ oder $A = g_\bot(a)$.
- Analog zu A gelte für $B = (\Sigma, Q_b, Q_{0,b}, \delta_b, F_b)$ entweder $B = g_\top(b)$ oder $B = g_\bot(b)$.
- Der Parameter $k \in \mathbb{N}$ sei die Parität des neuen Startzustands.
- Der Parameter $k' \in \mathbb{N}$ sei die Parität für alle Zustände aus dem verwendeten 2NFA $f_2(x)$.
- Der Parameter u sei ein Operator aus einer positiven booleschen Formel, also entweder $\wedge$ oder $\vee$. Weiter sei $\hat{u}$ gerade der entsprechend andere Operator als u.

Es ergibt sich der parametrisierte 2APW

$$(\Sigma, Q, Q_0, \delta, F)$$

mit

$$
\begin{aligned}
Q &= Q_a \cup Q_x \cup Q_b \cup \{q_0\} \\
Q_0 &= q_0 \\
\delta(q,s) &= \begin{cases}
Q_{0,b}\ u\ (Q_{0,a}\ \hat{u}\ \text{join}(u, Q_{0,x})) & \text{wenn } q = q_0 \wedge s = \varepsilon \\
\text{inject}(u, \delta_x(q,s), F_x, q_0) & \text{wenn } q \in Q_x \\
\delta_a(q,s) & \text{wenn } q \in Q_a \\
\delta_b(q,s) & \text{wenn } q \in Q_b
\end{cases} \\
F &= \{(q_0,k)\} \cup F_a \cup \{(q,k') \mid q \in Q_x\} \cup F_b.
\end{aligned}
$$

Abbildung 4.17 zeigt den parametrisierten 2APW für alle Power-Operatoren als schematischen Graphen.

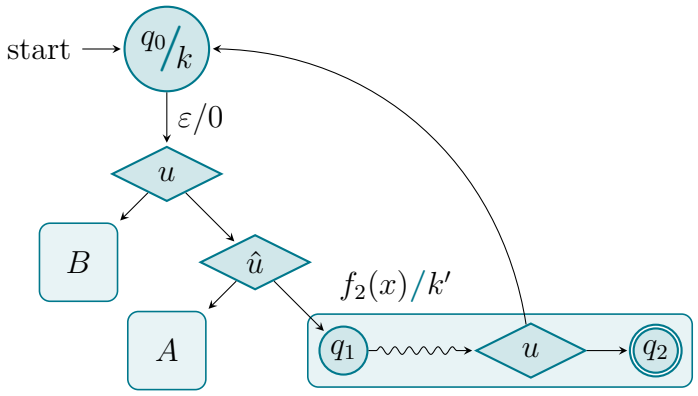

Abbildung 4.17.: Parametrisierter 2APW aller Power-Operatoren als Graph.

Tabelle 4.4 auf der nächsten Seite zeigt die Wahl der Parameter für die verschiedenen Power-Operatoren. Die Funktion g liefere für den entsprechenden Operator als Argument das gespiegelte 2APW-Paar bestehend aus dem Automaten, der sich mit den dort angegebenen Parametern für das Argument ergibt, und dem negierten Automaten, der sich mit den angegebenen Parametern für das negierte Argument ergibt.

Betrachten wir zum Beispiel den normalen Power-Operator $a/x\rangle\rangle b$, so besteht der nicht negierte 2APW für diesen Ausdruck aus den 2APWs für die Ausdrücke a und b und dem 2NFA für den Ausdruck x. Der Zustand q_0 erhält die Parität 1, sodass der große Zyklus irgendwann verlassen werden muss und dadurch der Rest des Wortes vom Versuch b akzeptiert werden muss. Die Zustände des 2NFAs von x erhalten ebenfalls alle die Parität 1, damit auch ein unendlich langes Verbleiben in diesem Teil des Automaten nicht zu einem akzeptierenden Lauf führt. Dieser Teil

Argument	A	B	k	k'	u	$\hat{u}$
$a/x\rangle\rangle b$	$g_\top(a)$	$g_\top(b)$	1	1	$\vee$	$\wedge$
$\neg(a/x\rangle\rangle b)$	$g_\bot(a)$	$g_\bot(b)$	0	0	$\wedge$	$\vee$
$a/x\rangle b$	$g_\top(a)$	$g_\top(b)$	2	1	$\vee$	$\wedge$
$\neg(a/x\rangle b)$	$g_\bot(a)$	$g_\bot(b)$	1	0	$\wedge$	$\vee$
$a//x\rangle\rangle b$	$g_\top(a)$	$g_\top(b)$	1	0	$\wedge$	$\vee$
$\neg(a//x\rangle\rangle b)$	$g_\bot(a)$	$g_\bot(b)$	2	1	$\vee$	$\wedge$
$a//x\rangle b$	$g_\top(a)$	$g_\top(b)$	0	0	$\wedge$	$\vee$
$\neg(a//x\rangle b)$	$g_\bot(a)$	$g_\bot(b)$	1	1	$\wedge$	$\vee$

Tabelle 4.4.: Wahl der Parameter für die verschiedenen Power-Operatoren.

des Automaten muss also über einen der akzeptierenden Zustände wieder verlassen werden. Der Operator u ist die Disjunktion $\vee$, sodass entweder der Automat B betreten wird und damit der Versuch b den Rest des Wortes akzeptiert, oder sich der Zyklus fortsetzt. Der Operator $\hat{u}$ ist die Konjunktion $\wedge$, sodass in diesem Fall der Automat A betreten wird, und somit die Pflicht a akzeptieren muss, und der Zyklus nach der Verzögerung x erneut beginnt.

4.3. Beispiel

In diesem Abschnitt wollen wir ein Beispiel für die obige Umwandlung betrachten. Es ist natürlich nicht möglich, ein Beispiel zu konstruieren, in dem alle Operatoren vorkommen. Trotzdem werden einige grundsätzliche Prinzipien an folgendem Beispiel deutlich werden.

Wir betrachten zunächst die LTL-Formel $\Box a$ über dem Alphabet $\Sigma = \{a, b\}$. Diese Formel prüft, ob in jedem Zeichen die Bedingung a gilt. Es werden also alle unendlichen Worte, die nur aus dem Zeichen a bestehen, akzeptiert. Wir wollen nun die Umwandlungsfunktion f_{LTL} aus Abschnitt 2.5 auf Seite 23 verwenden, um aus dieser LTL-Formel eine RLTL-Formel zu erzeugen. Es ergibt sich

$$f_{\mathrm{LTL}}(\Box a) = f_{\mathrm{LTL}}(a)/\operatorname{true}\rangle\emptyset = a; \neg\emptyset/\operatorname{true}\rangle\emptyset.$$

Diese RLTL-Formel prüft mit einer Verzögerung von einem beliebigen Zeichen, ob die Formel $a; \neg\emptyset$ gilt. Diese gilt gerade, wenn ein a und anschließend ein beliebiges Wort gelesen werden kann. Diese Formel akzeptiert also ebenfalls genau alle unendlichen Worte, bei denen jedes Zeichen ein a ist.

Aufbauend auf dieser Formel, wollen wir im Folgenden die Formel

$$\varphi := a; \neg\emptyset/\operatorname{true}\operatorname{true}\rangle\emptyset$$

ebenfalls über dem gleichen Alphabet betrachten. Im Gegensatz zur direkt aus der LTL-Formel $\Box a$ gewonnen Formel hat diese Formel eine Verzögerung true true, die zwei beliebige Zeichen akzeptiert. Diese RLTL-Formel prüft also für jedes zweite Zeichen, ob die Formel $a; \neg\emptyset$ erfüllt wird. Es werden also unendliche Worte über $\{a, b\}$ akzeptiert, bei denen jedes zweite Zeichen beginnend mit dem ersten Zeichen des Wortes ein a ist. Die vollen Möglichkeiten von RLTL würden wir erst ausschöpfen, wenn $a; \neg\emptyset$ kein einzelner Basisausdruck konkateniert mit der Pump-Konstruktion $\neg\emptyset$, die alles akzeptiert, sondern eine komplexere RLTL-Formel wäre. Das hier betrachtete Beispiel beschränkt sich aber auf diesen einfachen Fall, da das zugrundeliegende Prinzip auch hier deutlich wird.

Die Umwandlungsfunktion f aus diesem Kapitel würde rekursiv in die Formel absteigen. Wir beginnen aber mit den Abbrüchen dieser Rekursion und betrachten zunächst die Umwandlung des Basisausdrucks a. Entsprechend Unterabschnitt 4.1.1 auf Seite 41 ergibt sich

$$f_2(a) = (\Sigma, \{q_0, q_1\}, \{q_1\}, \{((q_0, a), (q_1, 1))\}).$$

Abbildung 4.18 zeigt den Graphen dieses Automaten. Er liest genau ein a und akzeptiert dann. Alle anderen Eingaben werden verworfen.

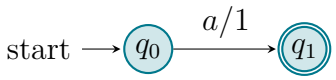

Abbildung 4.18.: $f_2(a)$ als Graph.

Um diesen Automaten in den zu erzeugenden 2APW einbauen zu können, muss er zu einem totalen Automaten erweitert werden (siehe Unterabschnitt 3.1.3 auf Seite 27). Dazu wird eine Senke q_2 hinzugefügt. Abbildung 4.19 zeigt den Graphen des totalen Automaten von $f_2(a)$.

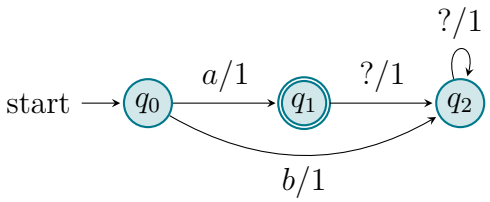

Abbildung 4.19.: Der totale Automat von $f_2(a)$ als Graph.

Der zweite Rekursionsabbruch findet sich bei der Umwandlung von true. true ist ebenfalls ein Basisausdruck und entsprechend Unterabschnitt 4.1.1 auf Seite 41 gilt hier

$$f_2(\text{true}) = (\Sigma, \{q_0, q_1\}, \{q_1\}, \{((q_0, ?), (q_1, 1))\}).$$

Abbildung 4.20 zeigt den Graphen dieses Automaten. Er akzeptiert alle Worte über dem Alphabet Σ der Länge genau 1. Alle anderen Eingaben werden verworfen.

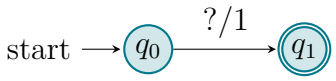

Abbildung 4.20.: $f_2(\text{true})$ als Graph.

Mit dem Ergebnis von $f_2(\text{true})$ können wir nun $f_2(\text{true}\,\text{true})$ bestimmen. Dabei werden entsprechend Unterabschnitt 4.1.3 auf Seite 42 zwei Automaten $f_2(\text{true})$ durch eine ε-Transition verbunden. Wir erhalten den Automaten, der als Graph in Abbildung 4.21 dargestellt ist. Dieser Automat akzeptiert nun alle Worte über dem Alphabet Σ, die genau die Länge 2 haben.

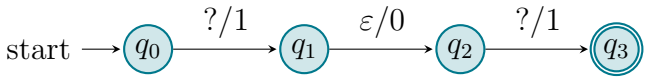

Abbildung 4.21.: $f_2(\text{true}\,\text{true})$ als Graph.

Der Automat $f_2(\text{true}\,\text{true})$ ist bereits fast total, da die ε-Transition unabhängig vom aktuell gelesenen Zeichen verwendet werden kann. Nur im letzten Zustand q_3 gibt es noch keine ausgehenden Transitionen. Der totale Automat mit entsprechend ergänzter Senke q_4 wird als Graph in Abbildung 4.22 dargestellt.

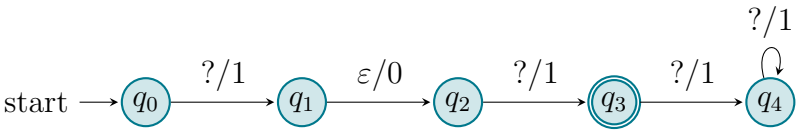

Abbildung 4.22.: Totaler Automat von $f_2(\text{true}\,\text{true})$ als Graph.

Um den Automaten $f_2(\text{true}\,\text{true})$ in einen 2APW einbauen zu können, muss dieser nicht nur wie im letzten Schritt zu einem totalen Automaten umgebaut werden, sondern die ε-Transition muss ebenfalls entfernt werden. Dabei wird der Zustand q_1 überflüssig und kann entfernt werden. Wir erhalten den Automaten, der als Graph in Abbildung 4.23 dargestellt wird.

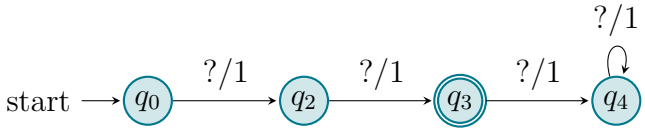

Abbildung 4.23.: Totaler Automat von $f_2(\text{true}\,\text{true})$ ohne ε-Transitionen als Graph.

Als letzten Rekursionsabbruch müssen wir nun noch die Umwandlung des RLTL-Ausdrucks betrachten. $g(\emptyset)$ ist dabei gerade das Paar von 2APWs, dass in Unterabschnitt 4.2.1 auf Seite 52 beschrieben wird und in Abbildung 4.12 auf Seite 53 als Paar von Graphen von dargestellt wird.

Mit dem totalen Automat von $f_2(a)$ und $g(\emptyset)$ können wir nun die Umwandlung von $a; \neg\emptyset$ betrachten. Entsprechend Unterabschnitt 4.2.3 auf Seite 54 liefert $g(a; \neg\emptyset)$ ein Paar von gespiegelten 2APWs. In $g_\top(a; \neg\emptyset)$ erhalten alle drei Zustände des 2NFAs $f_2(a)$ die Parität 1. Ein endloses Verweilen in der Senke q_2 führt also nicht zu einem akzeptierenden Lauf. Die Transition zum akzeptierenden Zustand q_1 wird erweitert zu einer Transition zu q_1 oder dem Startzustand von $g_\bot(\emptyset)$. Durch die Verwendung von $g_\bot(\emptyset)$ statt $g_\top(\emptyset)$ werden wir an dieser Stelle der Negation von $\emptyset$ in $a; \neg\emptyset$ gerecht. In $g_\bot(a; \emptyset)$ erhalten hingegen alle drei Zustände von $f_2(a)$ die Parität 0. Dieser Automat soll ja gerade das Komplement des ersten Automaten akzeptieren, sodass ein unendliches Verweilen in der Senke q_2 nun ein akzeptierender Lauf ist. Die Transition zu q_1 wird hier erweitert zu einer Transition zu q_1 und dem Startzustand von $g_\top(\emptyset)$. Das Paar $g(a; \neg\emptyset)$ besteht also aus einem Automaten, der alle Worte akzeptiert, die mit a beginnen, und einem Automaten, der alle anderen Worte – also alle Worte, die nicht mit a beginnen – akzeptiert. Bei dem hier betrachteten Alphabet Σ sind letztere gerade alle Worte, die mit b beginnen. Abbildung 4.24 auf der nächsten Seite zeigt das Paar von Graphen, das diesem Paar von Automaten entspricht.

An diesem Automaten kann man sehr deutlich die Bedeutung der generierten Senke erkennen. In $g_\top(a; \neg\emptyset)$ würde auch ohne q_2 funktionieren, da mit der Transition zu q_3 ein akzeptierender Lauf existiert. Aber in $g_\bot(a; \neg\emptyset)$ würde ohne q_2 und die Transition von q_0 nach q_2 kein akzeptierender Lauf existieren. Wir sehen an diesem Beispiel also, wie die Verwendung von totalen Automaten bei der Interpretation des 2NFA als 2UFA wichtig wird.

Im finalen Schritt wird nun aus dem totalen Automaten des 2NFA $f_2(\mathrm{true}\,\mathrm{true})$, dem 2APW $g(\emptyset)$ und dem 2APW $g(a; \neg\emptyset)$ die Übersetzung von $\varphi = a; \neg\emptyset / \mathrm{true}\,\mathrm{true}\rangle\emptyset$ zusammengesetzt. Da dies der letzte Schritt ist, werden wir nur noch $g_\top(\varphi)$ betrachten, da das zweite Element von $g(\varphi)$ nur für die weitere Verwendung von φ als Teil einer größeren Formel benötigt wird. Die Darstellung von $g_\top(\varphi)$ ist in Abbildung 4.25 auf Seite 65 zu sehen.

In $g_\top(\varphi)$ existiert ein zusätzlicher Startzustand q_0. Von diesem Zustand aus kann per ε-Transition entweder in den Startzustand des Versuchs $g(\emptyset)$ oder in den Starzustand der Verpflichtung $g(a; \neg\emptyset)$ und in den Startzustand der Verzögerung $f_2(\mathrm{true}\,\mathrm{true})$ übergegangen werden. Dort wo der 2NFA von $f_2(\mathrm{true}\,\mathrm{true})$ in den akzeptierenden Zustand wechselt ist nun auch ein Wechsel in den neuen Startzustand q_0 möglich. q_0 hat die Parität 2, sodass ein unendlich häufiger Besuch von q_0 zu einem akzeptierenden Lauf führt. Da $g(\emptyset)$ kein einziges Wort akzeptiert, ist dies auch die einzige

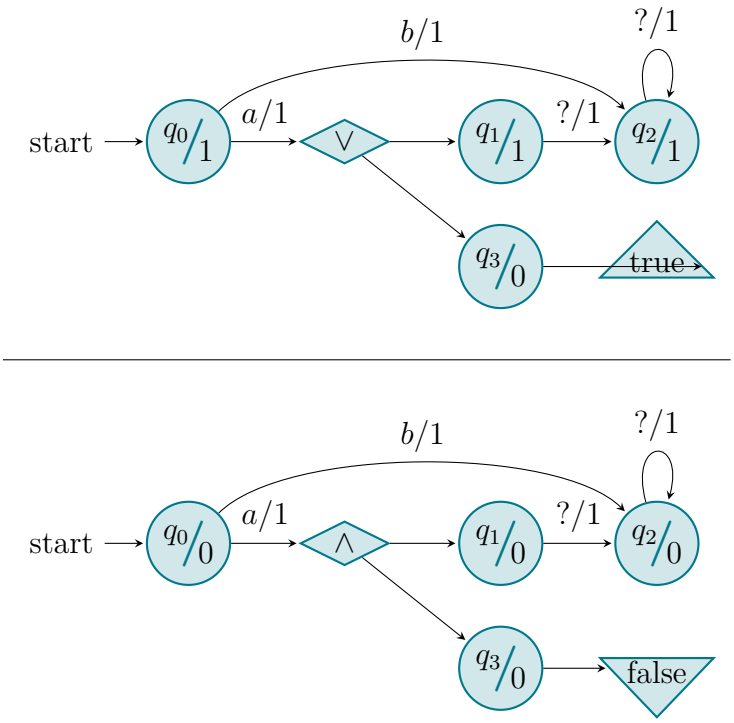

Abbildung 4.24.: Das Paar von Graphen, das dem 2APW-Paar $g(a; \neg\emptyset)$ entspricht.

Variante einen akzeptierenden Lauf zu erhalten. Es muss also alle zwei Zeichen der Eingabe immer wieder a gelten.

4.4. Optimierungen des erzeugten 2APWs

Betrachtet man das Ergebnis des Beispiels in Abbildung 4.25 auf der nächsten Seite, so erkennt man viele überflüssige Zustände und Transitionen. Beispielsweise der Übergang nach q_1 und von dort nach false kann zu einem Übergang nach false vereinfacht werden. Vereinfachungen dieser Art sind problemlos möglich, da bei Eingabeworten unendlicher Länge das false für jede Eingabe erreicht wird. Nach dem Streichen von q_1 wird an dieser Stelle nun die ganze Veroderung mit false nicht mehr benötigt. Auf ganz ähnliche Weise könnten in diesem Automaten viele Vereinfachungen vorgenommen werden. Die Möglichkeiten, den durch die Umwandlungsfunktion g entstandenen 2APW weiter zu vereinfachen, ohne eine vollständige Minimierung durchzuführen, wollen wir in diesem Abschnitt betrachten. Dabei geht es darum, ohne relevanten zusätzlichen Rechenaufwand möglichst viele Minimierungen vorzunehmen. Jeder Zustand, der bereits hier einfach entfernt werden konnte, kann die weitere Verarbeitung der Ausgabe vereinfachen. Bei den im Rahmen dieser Arbeit

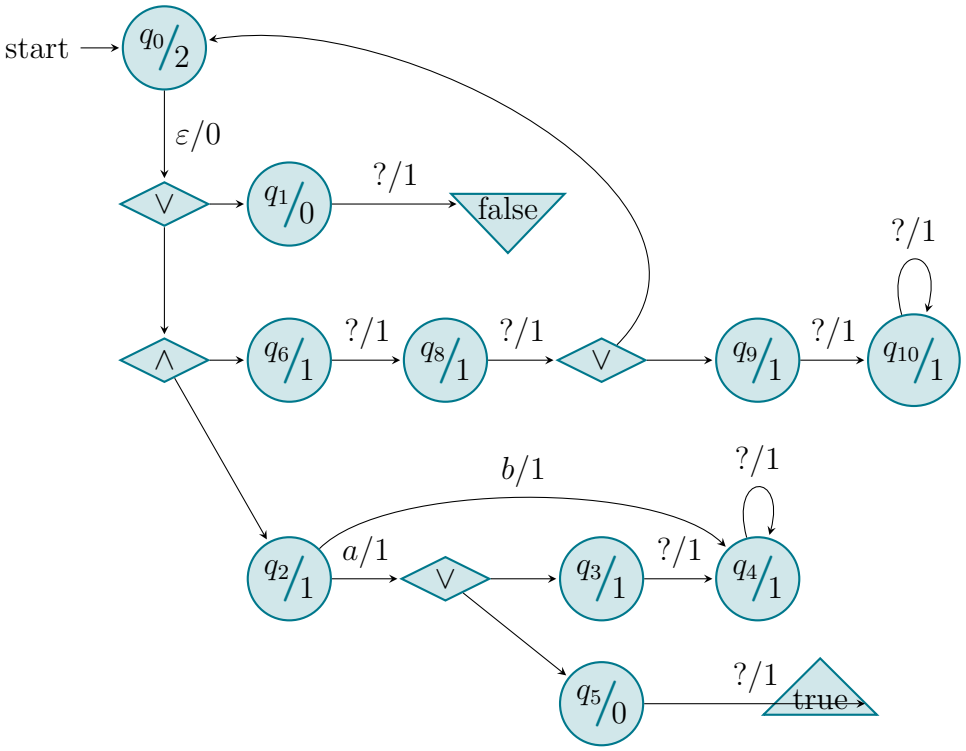

Abbildung 4.25.: $g_\top(a; \neg\emptyset/\,\mathrm{true}\,\mathrm{true}\rangle\emptyset)$ als Graph.

entwickelten Optimierungsverfahren soll aber kein minimaler Automat entstehen, denn eine vollständige Minimierung des Automaten (zum Beispiel in der Anzahl der Zustände) würde deutlich aufwändigere Ansätze erfordern.

Neben den möglichen Minimierungen des Automaten soll es auch um die Vereinfachung des Automaten gehen. Die Funktion g liefert einen 2APW mit ε-Transitionen. Für die weitere Verarbeitung dieses Automaten stellt sich die Frage, ob der Automat einfach in einen Ein-Wege-Automaten ohne ε-Transitionen umgewandelt werden kann.

4.4.1. Umwandlung des 2APWs in einen APW

Die Funktion g bildet auf die Menge der 2APWs ab, weil die verwendete Funktion f_2 zur Umwandlung von regulären Ausdrücken mit Vergangenheit auf die Menge der 2NFAs abbildet. Durch die Umwandlung der RLTL-Ausdrücke in alternierenden Paritätsautomaten selbst entstehen keine zusätzlichen Transitionen mit anderen Bewegungsrichtungen als $+1$. Auch die Funktion f_2 generiert nur für reguläre Ausdrücke mit Vergangenheit tatsächlich Transitionen mit anderen Bewegungsrichtungen. Für alle anderen regulären Ausdrücke werden nur Transitionen mit Bewegungsrichtung $+1$ generiert. Kommen in einem RLTL-Ausdruck also keine regulären Ausdrücke mit Vergangenheit vor, so entsteht ein 2APW, der direkt als APW interpretiert werden kann.

Entsteht aber ein echter 2APW mit verschiedenen Bewegungsrichtungen, so ist eine einfache Umwandlung in einen APW nicht möglich, da die Umwandlung von regulären Ausdrücken mit Vergangenheit durch f_2 in 2NFAs auch Automaten liefern kann, deren Semantik ohne Kontext nicht sinnvoll ist (vergleiche Unterabschnitt 4.1.5 auf Seite 43). Aus dieser Betrachtung folgt, dass der 2NFA nicht vor dem Einbau in den 2APW in einen NFA umgewandelt werden kann. Es muss also tatsächlich zunächst ein 2APW generiert werden und dieser anschließend in einen APW umgewandelt werden.

Für die Umwandlung von Zwei-Wege-Automaten in Ein-Wege-Automaten hat Shepherdson in [She59] einen konstruktiven Beweis über die Äquivalenz von Zwei-Wege- und Ein-Wege-Automaten gegeben. Der Beweis basiert indirekt auf Crossing Sequences. Dabei wird für jede Eingabe eine Funktion definiert, die für einen Startzustand angibt, in welchem Zustand sich der Automat befindet, wenn das Wort komplett gelesen wurde. Da der Automat aus einer endlichen Menge von Zuständen besteht, müssen dabei auch nur endlich viele Eingaben unterschieden werden. Grob gesagt bestehen die neuen Zustände dann aus diesen Funktionen. Im Prinzip wird durch diese Technik für jeden Zustand betrachtet, in welchem Zustand der Automat zurückkommen würde, wenn er sich von diesem Zustand aus nach links bewegt. Da

der Automat nur eine endliche Zustandsmenge hat, gibt es auch nur eine endliche Anzahl Möglichkeiten, in welche Zustände der Automat zurück kommen kann, wenn er nach links gegangen ist. Leider kann dieses Verfahren nicht auf die Umwandlung von 2APWs zu APWs angewendet werden, da aus dieser Umwandlung kein alternierender Automat entstehen kann. Für die Betrachtung, in welchem Zustand der Automat wieder ankommen würde, wenn er nach links geht, müsste das ganze Verhalten des alternierenden Automaten mit betrachtet werden. Im Ergebnis müsste die Anpassung dieses Verfahrens auf 2APWs eine direkte Umwandlung von 2APWs in Büchi-Automaten leisten.

Vardi schlägt in [Var89] basierend auf Shepherdsons Konstruktion ein neues Verfahren vor, bei dem aus einem nichtdeterministischen Zwei-Wege-Automaten direkt ein deterministischer Ein-Wege-Automat generiert werden kann. Das grundsätzliche Problem, dass Verfahren dieser Art nicht für die Umwandlung von 2APWs zu APWs verwendet werden können, trifft aber auch hier zu.

Da diese klassischen Verfahren zur Reduktion von Zwei-Wege-Automaten nicht auf die hier benötigte Umwandlung adaptiert werden können, würde es den Rahmen dieser Arbeit sprengen, die Umwandlung von 2APWs zu APWs zu realisieren. Aus diesem Grunde liefert die hier vorgestellte Umwandlung für RLTL-Ausdrücke ohne Vergangenheit einen APW und für RLTL-Ausdrücke mit Vergangenheit einen 2APW. Daher können nur in APWs umgewandelte RLTL-Ausdrücke ohne Vergangenheit direkt mit dem im Rahmen von [Sch11] entwickelten Programm weiter verarbeitet werden. Die bei der Umwandlung von RLTL mit Vergangenheit entstehenden 2APWs müssen vor der weiteren Umwandlung in NBWs zunächst in APWs umgewandelt werden. Diese Umwandlung wird weder in [Sch11] noch im Rahmen dieser Arbeit realisiert.

4.4.2. Entfernung der ε-Transitionen

Die Umwandlungsfunktion f liefert einen (2)APW mit ε-Transition. Im Gegensatz zu Transitionen mit anderen Bewegungsrichtungen als $+1$ entstehen die ε-Transitionen direkt in der Umwandlung von RLTL-Ausdrücken in (2)APWs unter Verwendung der NFAs, denn alle ε-Transitionen, die bei der Übersetzung der regulären Ausdrücke in (2)NFAs entstehen, müssen vor dem Einbau der (2)NFAs in die (2)APWs entfernt werden.

Um ε-Transitionen aus 2APWs zu entfernen, kann mit einigen wenigen Anpassungen das in Unterabschnitt 4.1.6 auf Seite 45 beschriebene Verfahren zum Entfernen von ε-Transitionen aus NFAs verwendet werden. Der grundsätzliche Unterschied besteht darin, dass die Ersetzungsfunktion t nicht mehr auf eine Menge von Paaren von Zuständen und Akzeptanzbedingungen abbildet, sondern auf eine positive boolesche

Formel $\mathcal{B}^+(Q \times \mathbb{N})$ von Paaren von Zuständen und Paritäten. Von dieser Funktion t kann nun ganz analog die Hülle gebildet werden, wobei jeweils das Maximum der alten Parität und der neuen Parität statt der Disjunktion aus der alten und der neuen Akzeptanzbedingung verwendet wird. Im dritten Schritt werden nun Paare (q, c) in den Funktionswerten von t_k gesucht, bei denen $c > F(q)$ gilt. Entsprechend den zwei Fällen wird dann entweder die Parität des Zustandes q angepasst oder ein neuer Zustand q' mit der Parität c erzeugt. Im vierten und fünften Schritt wird die Ersetzungsfunktion nun auf die Transitionsfunktion δ und die initiale Formel Q_0 angewendet. Die entstandenen nicht erreichbaren Zustände können am Ende durch die ebenfalls in Unterabschnitt 4.1.6 auf Seite 45 bereits beschriebene Tiefensuche entfernt werden.

Das ganze Verfahren kann genauso auf 2APWs angewendet werden. Die Ersetzungsfunktion t bildet noch immer nur auf positive boolesche Formeln von Paaren von Zuständen und Paritäten ab, da ε-Transitionen nur die Bewegungsrichtung 0 haben können. Die einzige Anpassung des Verfahrens muss im vierten Schritt bei der Anwendung der Ersetzungsfunktion t_k auf die Transitionsfunktion δ erfolgen. Hier muss beim 2APW die Bewegungsrichtung der ursprünglichen Transition erhalten werden.

Bei der hier vorgestellten Entfernung von ε-Transitionen wird der Automat nicht zwingend übersichtlicher, da zwar die Anzahl der Zustände reduziert wird, aber die Anzahl an komplizierten Transitionen steigt. Die Optimierung des Beispielautomaten in Abbildung 4.25 auf Seite 65 in Unterabschnitt 4.4.5 auf Seite 71 illustrieren diesen Effekt. Durch das Ersetzen von Zustand q_0 durch die ausgehende Transition dieses Zustands kommt diese komplexe Transition nun zweimal vor: Einmal als initiale Zustandsformel Q_0 und einmal als Transition vom Zustand q_2 aus. In diesem kleinen Beispiel führt dies noch nicht zu Problemen, aber bei größeren Automaten kann durch diesen Effekt die Entfernung von ε-Transitionen schnell zu unübersichtlichen Automaten führen.

Die Entfernung von ε-Transitionen ist aber insbesondere für die weitere Verarbeitung des APWs durch das im Rahmen von [Sch11] entwickelte Programm sehr wichtig, da der in dieser Arbeit verwendete Algorithmus APWs mit ε-Transitionen nur unter sehr eingeschränkten Randbedingungen verarbeiten kann. Solange auf allen Pfaden eines Laufs gleichzeitig eine ε-Transition gewählt wird, kann das ε einfach als zusätzliches Zeichen des Eingabealphabets gesehen werden. ε-Transitionen können aber auf verschiedenen Pfaden unabhängig gewählt werden, da der APW nach dem Verwenden der ε-Transition noch immer das gleiche Eingabezeichen liest wie vorher. Aus diesem Grunde kann ε nicht einfach als Eingabezeichen modelliert werden, sondern muss analog zur Transition mit der Bewegungsrichtung 0 im Zwei-Wege-Automaten gesondert behandelt werden.

4.4.3. Entfernung nicht benötigter Transitionen

Wir haben bereits bei der Analyse des Beispiels in Abbildung 4.25 auf Seite 65 gesehen, dass einige Transitionen trivial entfernt werden können. Allgemein betrifft das alle Transitionen, die unabhängig von der weiteren Eingabe zu true oder false führen. Um dieses Problem allgemeingültig zu lösen, müssten alle Zyklen im Graphen des Automaten gefunden werden, die unabhängig von der konkreten Eingabe immer zum Verwerfen oder zum Akzeptieren der Eingabe führen. In dieser Arbeit werden aber nur die Fälle betrachtet, die in der hier verwendeten Umwandlung häufig auftreten. Anstelle von komplexen Zyklen entstehen hier einzelne Zustände, die unabhängig von der Eingabe nicht mehr verlassen werden können. Zustände solcher Art entstehen aus den Senken, die eingeführt werden, um die NFAs der regulären Ausdrücke in totale Automaten zu verwandeln. Eine solche Senke ist immer der einzige Zustand einer Spur, der unendlich oft besucht wird, wenn eine Spur einen solchen Zustand betritt, da der Weg in diesen Zustand ein endlicher Weg sein muss. Aus diesem Grunde ist das Maximum der Paritäten aller unendlich oft besuchten Zustände immer gerade die Parität dieses Zustandes. Hat eine solche Senke also eine gerade Parität (0 oder 2), so kann sie durch true ersetzt werden, denn das minimale Modell von true ist gerade die leere Menge {}, sodass eine Spur, die mit einer Transition zu true endet, auf jeden Fall akzeptierend ist. Hat eine solche Senke die ungerade Parität 1, so kann sie durch false ersetzt werden, denn es existiert kein minimales Modell von false, sodass keine akzeptierende Spur existieren kann, die eine Transition zu false enthält.

Im nächsten Schritt betrachten wir nun Zustände, aus denen die Transitionsfunktion für jede Eingabe immer true bzw. immer false liefert. Dabei ist zu beachten, dass bei der partiellen Transitionsfunktion nicht angegebene Transitionen für eine Eingabe in einem Zustand als false gelten. Ein solcher Zustand kann nun wiederum durch true bzw. false ersetzt werden. Diese Betrachtung kann rekursiv fortgesetzt werden, indem für jede positive boolesche Formel einer Transitionsfunktion für jeden Zustand betrachtet wird, ob dieser nur zu true oder nur zu false auswertet. Entsprechend den Regeln für boolesche Ausdrücke können die Vorkommen von true und false dann soweit reduziert werden, bis true und false nur noch als ganze Formeln auftreten. Dabei werden die folgenden Äquivalenzen für eine beliebige positive boolesche Formel φ zur Reduktion der Formel verwendet:

- $\text{true} \wedge \text{true} \equiv \text{true}$
- $\text{false} \wedge \varphi \equiv \text{false}$
- $\varphi \wedge \text{false} \equiv \text{false}$
- $\text{true} \wedge \varphi \equiv \varphi$
- $\varphi \wedge \text{true} \equiv \varphi$
- $\text{false} \vee \text{false} \equiv \text{false}$

- $\text{true} \vee \varphi \equiv \text{true}$
- $\varphi \vee \text{true} \equiv \text{true}$
- $\text{false} \vee \varphi \equiv \varphi$
- $\varphi \vee \text{false} \equiv \varphi$

Diese rekursive Betrachtung beginnt mit der initialen Zustandsformel Q_0 und wertet rekursiv die Formel aus, um dann rekursiv jeden Zustand auszuwerten, indem wieder rekursiv jede Formel aller ausgehenden Transitionen ausgewertet wird. Dabei muss die Rekursion abbrechen, wenn ein Zustand bereits besucht wurde, damit Zyklen nicht zu Endlosschleifen führen. Aus diesem Grunde können Zyklen durch dieses einfache Verfahren auch nicht korrekt reduziert werden, auch wenn die ganze Schleife nur zu true oder false auswertet.

Wie beim Entfernung von ε-Transitionen erzeugt dieses Verfahren Zustände, die nicht mehr erreichbar sind. Diese Zustandsreste können durch die gleiche in Unterabschnitt 4.1.6 auf Seite 45 bereits beschriebene Tiefensuche zur Markierung aller erreichbaren Knoten wie bei der Entfernung der ε-Transitionen entfernt werden.

Entfernen von false und Ersetzen von true

Nach dem Entfernen nicht benötigter Zustände wurden alle Transitionen so angepasst, dass true und false nur noch als ganze Formeln vorkommen können. Für die weitere Verarbeitung ist es unter Umständen einfacher, wenn die verwendeten positiven booleschen Formeln frei von true und false sind. Insbesondere ist dies eine Voraussetzung für die Verwendung des im Rahmen von [Sch11] entwickelten Programms. Aus diesem Grunde werden alle Transitionen zu false direkt entfernt, da alle nicht definierten Übergänge der partiellen Transitionsfunktion gerade als false interpretiert werden können. Auf diese Weise wird der Automat auch unabhängig von der weiteren Verarbeitung übersichtlicher, da auf diese Weise gerade bei größeren Eingabealphabeten viele Transitionen wegfallen.

Alle Transitionen zu true können durch eine Transition zu einem einzigen neuen Zustand ersetzt werden. Dieser Zustand hat Parität 0 oder 2 und für jedes Zeichen des Eingabealphabets eine Transition zu sich selbst mit der Bewegungsrichtung +1. Mit dem gleichen Argument wie beim Entfernen nicht benötigter Transitionen in Unterabschnitt 4.4.3 auf der vorherigen Seite entspricht ein solcher Zustand gerade true.

4.4.4. Zusammenfassung

Für die praktische Verwendung der Optimierungsschritte ist es sinnvoll, zwei verschiedene Optimierungen anzubieten. Um einen möglichst übersichtlichen Automaten für die Verwendung durch Menschen zu erzeugen, werden

– nicht benötigten Transitionen entfernt,
– und alle Vorkommen von false gelöscht.

Für eine möglichst einfache automatische Weiterverarbeitung des Automaten werden

– nicht benötigten Transitionen entfernt,
– alle ε-Transitionen entfernt,
– alle Vorkommen von false gelöscht,
– und alle Vorkommen von true durch einen neuen Zustand ersetzt.

4.4.5. Optimierung des Beispiels aus Abschnitt 4.3

In diesem Unterabschnitt wollen wir das bereits in Abschnitt 4.3 auf Seite 60 betrachtete Beispiel weiter untersuchen und die beiden verschiedenen Verfahren zur Optimierung des 2APWs anwenden. Wir beginnen damit, alle nicht benötigten Transitionen zu entfernen und alle Vorkommen von false zu löschen. Auf diese Weise entsteht der Automat, der in Abbildung 4.26 auf der nächsten Seite als Graph repräsentiert wird. Im Vergleich zu dem in Abbildung 4.25 auf Seite 65 repräsentierten Graphen ist eine deutliche Einsparung an Zuständen zu erkennen. So wurde zum Beispiel der Zustand q_1 durch false ersetzt und damit die ganze Veroderung mit false an der Stelle entfernt. Entsprechend wurde auch die nie zu einem akzeptierenden Pfad führenden Zweige nach q_8 und q_2 entfernt. Neben dieser leichten Optimierung können wir das Ergebnis der starken Optimierung in Abbildung 4.27 auf der nächsten Seite betrachten. Hier wurden zusätzlich alle ε-Transitionen entfernt, sodass der Zustand q_0 entfernt wurde und das eine Vorkommen von true wurde durch einen neuen Zustand ersetzt, der immer einen akzeptierenden Pfad generiert. Als Ergebnis erhalten wir einen Automaten, der zwar einfacher weiter verarbeitet werden kann, da er keine Transitionen zu true oder false und keine ε-Transitionen enthält, aber dafür auch schwerer zu lesen ist. Aus dem einem $\wedge$ in Abbildung 4.26 auf der nächsten Seite wurden zwei $\wedge$, um die ε-Transition vorweg zu nehmen, und auch die Bedeutung von q_9 erschließt sich nicht mehr auf den ersten Blick. Es muss je nach weiterer Verwendung die beste Optimierung ausgewählt werden.

Abschließend bleibt noch zu vermerken, dass der hier erzeugte 2APW auch direkt als APW interpretiert werden kann, da alle Transitionen die Bewegungsrichtung

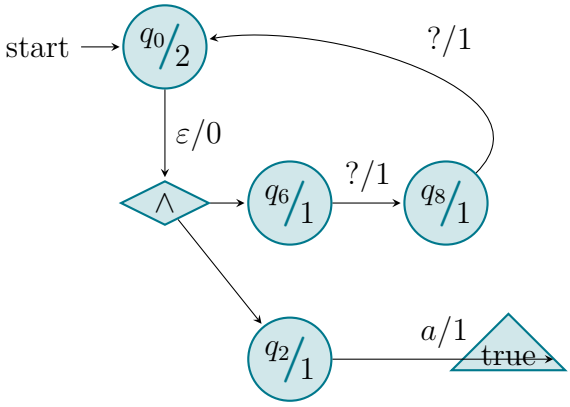

Abbildung 4.26.: $g_\top(a; \neg\emptyset/\,\mathrm{true}\,\mathrm{true}\rangle\emptyset)$ mit leichter Optimierung als Graph.

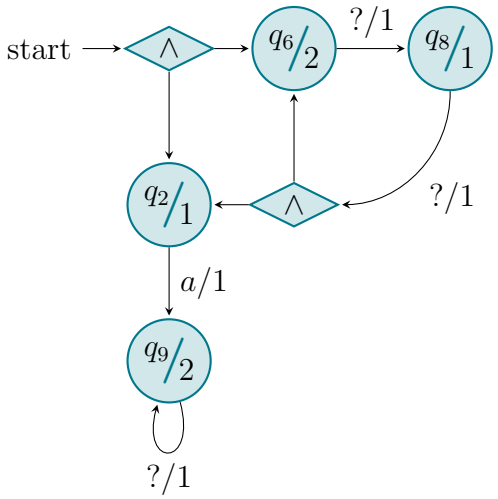

Abbildung 4.27.: $g_\top(a; \neg\emptyset/\,\mathrm{true}\,\mathrm{true}\rangle\emptyset)$ mit starker Optimierung als Graph.

+1 haben. In der im folgenden Kapitel beschriebenen praktischen Implementierung des hier vorgestellten Verfahrens findet genau diese Vereinfachung automatisch statt. Echte Zwei-Wege-Automaten entstehen nur bei der Umwandlung von RLTL-Ausdrücken mit Vergangenheit aus der Umwandlung der dort verwendeten regulären Ausdrücke mit Vergangenheit in 2NFAs.

5. Implementierung

In diesem Kapitel werden die Details der Implementierung, die im Rahmen dieser Arbeit erstellt wurde, beschrieben. In der Implementierung wurden alle bisher erläuterten Umwandlungsverfahren und Optimierungsstrategien realisiert. Dabei wurde insbesondere Wert auf die Erstellung einer wiederverwendbaren Bibliothek gelegt, deren einzelne Module auch in anderen Kontexten verwendet werden können.

Zur einfachen Verwendung der Bibliothek existiert eine Kommandozeilenanwendung, mit der die einzelnen Schritte der Umwandlung ausgeführt werden können. Neben der eigentlichen Umwandlung von RLTL-Formeln in alternierende Paritätsautomaten können damit die einzelnen Schritte der Umwandlung überdies separat verwendet werden. Unter anderem können zum Beispiel alternierende Paritätsautomaten eingelesen werden und auf diese die Optimierungsschritte angewendet werden. Weiter ist es möglich, reguläre Ausdrücke mit Vergangenheit in 2NFAs umzuwandeln, ε-Transitionen aus 2NFAs und 2APWs zu entfernen und ω-reguläre Ausdrücke in RLTL-Ausdrücke umzuwandeln. Schließlich wurde analog zum RLTL-Parser ein Parser für LTL-Ausdrücke realisiert, sodass LTL-Ausdrücke in RLTL-Ausdrücke umgewandelt werden können.

Für die Implementierung wurde die Programmiersprache Scala verwendet. Diese relativ neue Programmiersprache kombiniert Ideen aus der objektorientierten und der funktionalen Programmierung. Im Gegensatz zu anderen Neuschöpfungen von Programmiersprachen wird Scala für die Java Virtual Machine kompiliert und ist vollständig kompatibel zu Java, sodass Java-Klassen in Scala und umgekehrt direkt verwendet werden können. Aus diesem Grunde kann die im Rahmen dieser Arbeit erstellte Bibliothek als JAR-Archiv in einem beliebigen Java-Projekt eingebunden werden. Durch die Wahl von Scala war es möglich, Konzepte moderner Programmiersprachen mit einer maximalen Kompatibilität mit existierenden Systemen zu kombinieren.

Scala besitzt neben den aus Java bekannten Sprachkonstrukten einige neue Elemente der objektorientierten Programmierung. So bieten Traits die Möglichkeit, Interfaces um Implementierungen zu erweitern. Ein Trait kann allerdings im Gegensatz zu einer Klasse nicht direkt instanziiert werden, sodass mit Traits zwar Mehrfachvererbungen möglich sind, aber einige Probleme der Mehrfachvererbung, die zum Beispiel aus C++ bekannt sind, vermieden werden. Singleton-Objekte sind ebenfalls eine Besonderheit in Scala: Von einer Klasse, die als Singleton-Objekt gekennzeichnet ist,

existiert stets nur eine Instanz, die nicht erzeugt werden muss, sondern direkt zur Verfügung steht. Trägt ein Singleton-Objekt den gleichen Namen wie eine Klasse im gleichen Paket, so wird es als Kompagnonobjekt bezeichnet und kann unter anderem Factory-Methoden, Operatoren und Hilfsmethoden für das Pattern-Matching für seine Klasse zur Verfügung stellen.

Darüber hinaus bietet Scala die Möglichkeit, das aus funktionalen Sprachen bekannten Pattern von unveränderlichen Variablen mit objektorientierten Klassen zu kombinieren. Dabei entstehen unveränderliche Klassen, die effizient kopiert und auf diesem Wege manipuliert werden können. Der Vorteil dieser Technik liegt darin, dass, wie in funktionalen Sprachen üblich, frei von Seiteneffekten programmiert werden kann. Der Aufruf einer Methode kann eine solche Klasse nicht verändern, sodass das Ergebnis eines Methodenaufrufs nur im Rückgabewert der Methode liegen kann. Dadurch kann auf Getter- und Setter-Methoden vollständig verzichtet werden und Eigenschaften von Klassen können direkt öffentlich zur Verfügung gestellt werden, da diese Variablen durch Nutzer der Klasse nicht verändert werden können. Um trotzdem flexibel in der Implementierung zu bleiben, können Variablen und Methoden des gleichen Rückgabetyps transparent gegeneinander ausgetauscht werden, ohne dass sich die Schnittstelle ändert. Diese Technik der immutable case classes eignet sich sehr gut, um ein Umwandlungsverfahren mit Bottom-Up-Konstruktion zu realisieren, bei dem rekursiv in die eine Datenstruktur abgestiegen wird, um daraus die andere Datenstruktur zusammen zu setzen. Konkret wird auf diesem Wege eine RLTL-Formel durch die Anwendung von Pattern-Matching auf immutable case classes rekursiv in ihre einzelne Bestandteile zerlegt und diese werden zu alternierenden Paritätsautomaten umgewandelt, die dann wieder zu größeren Automaten zusammengesetzt werden.

Ein weiterer Vorteil der funktionalen Aspekte Scalas liegt in den vorhandenen Datenstrukturen, die in Datenstrukturen für Formeln oder Automaten verwendet werden können. Wie aus anderen funktionalen Programmiersprachen wie Erlang oder Haskell bekannt, können auch in Scala sehr leicht primitive Datentypen zu Listen oder Tupeln zusammengesetzt werden und es steht eine große Menge an Methoden zur Manipulation von Listen zur Verfügung. Auf diese Weise können zum Beispiel Automaten sehr dicht an der mathematischen Modellierung als 5-Tupel dargestellt werden. Dieses besteht dabei aus einer Liste von Zeichen (dem Eingabealphabet), einer Liste von Zuständen, einer Liste von Startzuständen, einer Zuordnung von Paaren von Zuständen und Zeichen auf eine Liste von Zuständen (der Transitionsfunktion) und einer Liste der akzeptierenden Zustände. Angenommen es existiert eine Klasse `Sign` für Zeichen des Eingabealphabets und eine Klasse `State` für Zustände, so könnte ein Typ für NFAs in Scala definiert werden als

```
Tuple5[Set[Sign], // Eingabealphabet
  List[State],    // Zustaende
```

```
  List[State],      // Startzustaende
  Map[Pair[State, Sign], List[State]],
                    // Transitionsfunktion
  List[State]]      // akzeptierende Zustaende
```

Dabei wird statt der Klasse `Set`, der direkten Entsprechung einer mathematischen Menge, die Klasse `List` verwendet. Im wesentlichen unterscheiden sich Listen und Mengen dadurch, dass Listen eine Reihenfolge speichern und Elemente mehrfach vorkommen können. Listen können in Scala deutlich effizienter als Mengen realisiert werden, da Mengen permanent die Eigenschaft, dass jedes Element nur einmal vorkommt, sicherstellen müssen. Diese Bedingung kann in diesem Fall aber sehr gut manuell sichergestellt werden, da sie nur an wenigen Stellen verletzt werden kann.

In dieser Arbeit wird statt des obigen Tupels eine etwas andere Modellierung über eine eigene Klasse `Nfa` verwendet, sodass eigene Methoden auf der Klasse definiert werden können. Die Eigenschaften `alphabet`, `states`, `start`, `transitions` und `accepting` werden aber ganz analog zu den Elementen des Tupels definiert.

Bei der Implementierung ging es nicht primär um eine möglichst performante Realisierung der Verfahren, sondern darum, eine möglichst gut wieder zu verwendende und nachvollziehbare Realisierung zu schaffen. Auch wenn das nicht zwingend ein Gegensatz sein muss, wäre es sicherlich möglich gewesen, in anderen Programmiersprachen und unter Verwendung anderer softwaretechnischer Paradigmen effizientere Implementierungen zu erzeugen. Der Schwerpunkt wurde aber bewusst auf eine elegante, gut wartbare, leicht zu verwendende und für andere Projekte einfach wiederzuverwendende Realisierung der Verfahren gelegt.

Als IDE und als Build-Tool wurde Eclipse verwendet, sodass der Quellcode als Eclipse-3.7-Projekt vorliegt. Im Ordner `src` befindet sich der eigentliche Quellcode, im Ordner `bin` befinden sich die kompilierten Klassen und im Ordner `test` befinden sich einige Unit-Tests, die allerdings bei weitem nicht den ganzen Funktionsumfang der Bibliothek abdecken. Schließlich befindet sich im Ordner `runnable` das exportierte JAR-Archiv und ein Windows-Batch-Script, das den Start des Programms vereinfacht. Im Unterordner `dwyer` befinden sich einige Windows-Batch-Scripte und Eingabedaten, mit denen die Implementierung auf praktisch relevanten Eingaben getestet wurde (siehe Abschnitt 5.8 auf Seite 109).

5.1. Scala und UML

Die Darstellung von Scala-Klassen in den folgenden UML-Diagrammen orientiert sich an [Rac09, Anhang C: UML-Erweiterungen für Scala]. Kurz zusammengefasst schlägt Rachimow folgende Konventionen vor:

- Für alle Attribute, die nicht als privat gekennzeichnet sind, existieren implizite Getter-Methoden.
- Für generische Klassen wird, wie in UML üblich, ein abstrakter Typ angegeben. Für diesen abstrakten Typ wird allerdings die Scala-Notation verwendet.
- Traits werden als abstrakte Klassen mit dem Stereotyp «trait» dargestellt. Die Vererbung von Traits untereinander und das Einmischen von Traits in Klassen wird mit dem gestrichelten Vererbungspfeil dargestellt. Letzteres wird mit dem Stereotyp «mixin» gekennzeichnet.
- Singleton-Objekte werden als normale Klassen dargestellt, aber darüber hinaus mit dem Stereotyp «singleton» gekennzeichnet. Da ein Kompagnonobjekt den gleichen Namen wie die zugehörige Klassen hat, können dabei zwei Klassen gleichen Namens entstehen. Dies ist jedoch nur erlaubt, wenn es sich wirklich um das zugehörige Kompagnonobjekt handelt, was durch einen Abhängigkeitspfeil mit dem Stereotyp «hasA» gekennzeichnet wird.

5.2. Paketstruktur

Das Projekt gliedert sich im Wesentlichen in drei Bereiche: die Automaten, die Formeln und die Kommandozeilenanwendung. Letztere befindet sich im Paket `cli` und verwendet die Klassen der anderen Pakete wie ein normaler Nutzer der Bibliothek. Im Paket `automata` befinden sich – wie in Abbildung 5.1 auf der nächsten Seite zu sehen – die abstrakte Klasse `Automata` und die erbenden Klassen `Apw` und `Nfa`, sowie die von beiden Automaten genutzten Klassen. Dazu gehören die Klasse `Direction` mit den erbenden Singleton-Objekte `Back`, `Pause` und `Forward` und die Klasse `Sign` mit dem erbenden Singleton-Objekt `All`, das als Platzhalter für ein beliebiges Zeichen verwendet wird, und der erbenden Klasse `Text`. Mit dieser Klasse werden Zeichen des Eingabealphabets der Automaten repräsentiert, wobei ein Zeichen der Eingabe durchaus durch einen mehrere Zeichen langen String repräsentiert werden kann. Auf diese Weise können Ereignisse mit sprechenden Namen als Zeichen des Alphabets verwendet werden. Das Eingabealphabet selbst ist als `Set[Sign]` definiert. Ebenfalls bei den gemeinsam genutzten Klassen befindet sich die Klasse `State`. Für die alternierenden Automaten werden noch die Klassen `ApwSpecular` zur Speicherung von gespiegelten Automatenpaaren und die generische Klasse `PosBool` zur Repräsentation von positiven booleschen Formeln über beliebigen anderen Klassen benötigt. Schließlich befinden sich die Parser-Klassen für Automaten ebenfalls in diesem Paket.

Wie man in Abbildung 5.2 auf Seite 79 erkennen kann, befindet sich im Paket `formula` nur die Klasse `Formula`, die eine Formel und ein Alphabet speichert,

die abstrakte Klasse `Expression` und die Klasse Alphabet. Dabei ist die Klasse Alphabet in der Abbildung aber nur als Platzhalter zu verstehen, da tatsächlich von Formeln das gleiche Alphabet wie von Automaten verwendet wird, sodass Alphabete eigentlich vom Typ `Set[automata.Sign]` sind. Neben diesem Paket `formula` existiert für jede Art unterstützter Formeln ein eigenes Paket, in dem sich jeweils eine eigene abstrakte Klasse `Expression` befindet, die von `formula.Expression` erbt. Von dieser Klasse erben dann alle anderen Klassen, die Operatoren und Konstanten dieser Formeln repräsentierten. Auf diese Weise werden die Klasse `ltl.Expression`, die Klasse `omegaregex.Expression`, die Klasse `regex.Expression` und die Klasse `rltl.Expression` definiert. Im Gegensatz zu Automaten besitzen Formeln selbst kein Alphabet, sondern erst in der Klasse `Formula` wird für die weitere Verwendung eine Formel und ein Alphabet zusammengeführt. Daher muss bei der Umwandlung eines Ausdrucks (`Expression`) in einen Automaten immer von außen eine Formel mitgegeben werden. Diese unterschiedliche Speicherung der Alphabete ist notwendig, da bei den Formeln im Gegensatz zu den Automaten keine geschlossene Einheit existiert. Jede Teilformel ist in sich eine vollständige Formel und die oberste Einheit der Formel ist nicht weiter als Wurzel-Element ausgezeichnet, sondern kann selbst unmittelbar als Teil einer anderen Formel verwendet werden. Ebenfalls im jeweiligen Paket der Formeln befinden sich auch die Parser-Klassen für die Formeln. Die Parser-Klassen wurden allerdings in Abbildung 5.1 und Abbildung 5.2 weggelassen, da diese gesondert in Abschnitt 5.4 auf Seite 85 bzw. Unterabschnitt 5.5.5 auf Seite 99 betrachtet werden.

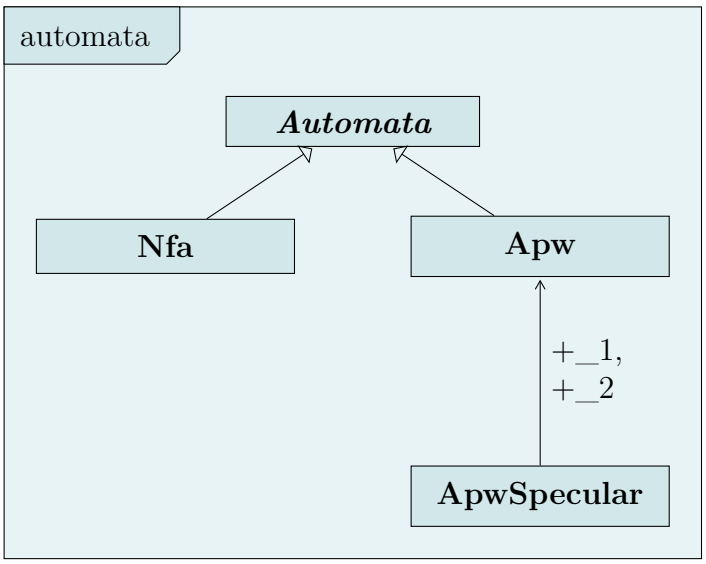

Abbildung 5.1.: UML-Klassendiagramm der Struktur der Automaten-Klassen im Paket `automata`.

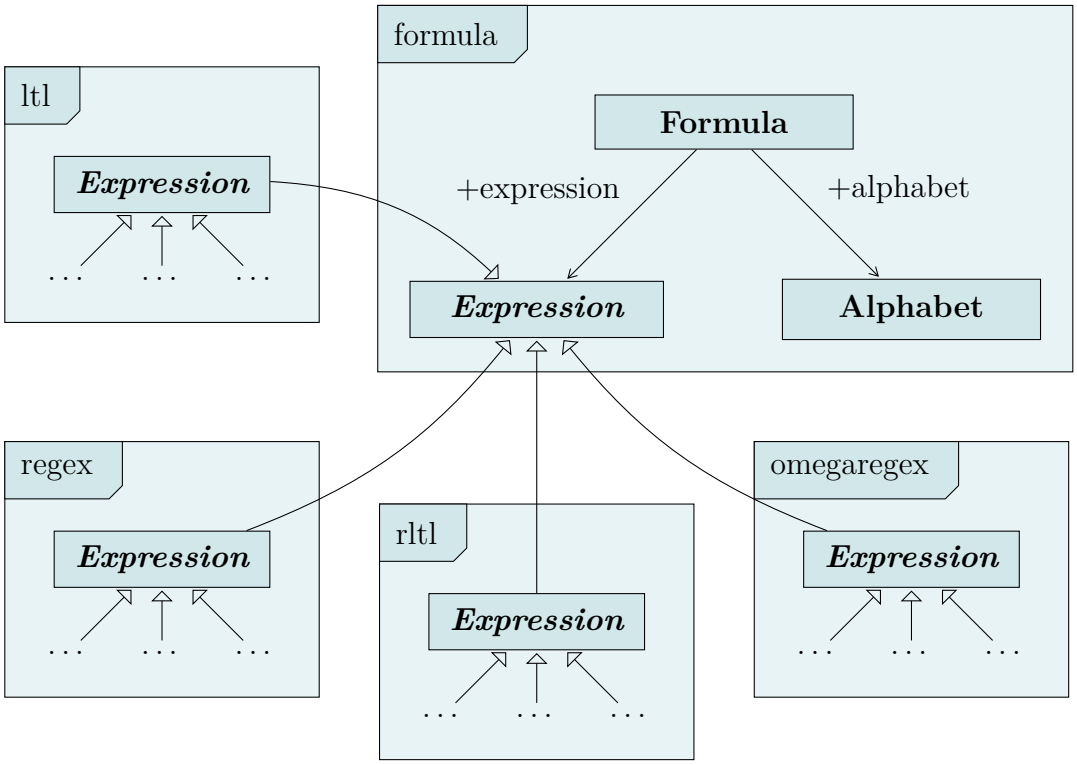

Abbildung 5.2.: UML-Klassendiagramm der Struktur der Logik-Klassen in den Paketen `formula`, `ltl`, `rltl`, `regex` und `omegaregex`.

5.3. Formeln

Formeln werden, wie im vorherigen Abschnitt bereits erläutert, als Instanzen der Klasse `Formula` gespeichert. Dabei besteht jede Formel aus einem Ausdruck und einem Alphabet. Ein solcher Ausdruck ist nun entweder ein Operator, der wiederum Ausdrücke enthält, ein Basisausdruck oder eine andere Konstante, wie die leere Sprache bei RLTL-Formeln. Wir wollen diese Struktur von Klassen am Beispiel von RLTL näher betrachten. Abbildung 5.3 auf Seite 82 zeigt alle Klassen, die zur Repräsentation einer RLTL-Formel benötigt werden. Alle diese Klassen erben von der abstrakten Klasse `Expression` aus dem Paket `rltl`. Für verschiedene weitere Verarbeitungsschritte – unter anderem für die textuelle Ausgabe von RLTL-Formeln – ist eine weitere Strukturierung nach der Operator-Rangfolge sinnvoll. Deswegen existieren die abstrakten Klassen `ConDisjunction`, `Clause`, `Entity` und `Operation`, die alle von `Expression` erben. Alle konkreten Klassen, die von einer solchen Klassen erben, entsprechen Operatoren (oder Konstanten) einer Ebene in der Rangfolge der Operatoren. Die Klasse `Empty` repräsentiert die einzige Konstante in RLTL-Formeln, da Basisausdrücke über reguläre Ausdrücke in RLTL-Formeln eingebunden werden. Entsprechend ist die Klasse `Empty` als Singleton-Objekt in Scala deklariert, sodass stets nur eine einzige Instanz zur Verfügung steht. Alle anderen Klassen enthalten wiederum mindestens ein Attribut vom Typ `Expression`. Alle Klassen, die von `Entity` oder von `Operation` erben, enthalten darüber hinaus ein Attribut vom Typ `Expression` aus dem Paket `regex`. Diese Operatoren verbinden einen Operanden aus der Menge der regulären Ausdrücke mit Operanden aus der Menge der RLTL-Ausdrücke. Durch die Verwendung der abstrakten Oberklasse aus dem Paket der regulären Ausdrücke kann diese Beziehung direkt als Attribut nachgebildet werden.

Auch wenn die Operatoren einer gemeinsamen Oberklasse in der Regel über die gleichen Attribute zur Speicherung der Operanden verfügen, werden diese trotzdem erst in den konkreten Klassen definiert. Da in den gemeinsamen abstrakten Klassen keine Methoden existieren, die diese Attribute benutzen, entstehen durch dieses Vorgehen keine Nachteile. Dafür kann Scalas impliziete Generierung von Getter-Methoden, Vergleichs-Methoden und Hilfsmethoden für das Pattern-Matching auf diese Weise optimal genutzt werden.

RLTL-Formeln enthalten selbst keine Basisausdrücke. Erst in den verwendeten regulären Ausdrücken sind Basisausdrücke, und damit Elemente des Eingabealphabets, enthalten. Diese werden durch die Klasse `Letter` aus dem Paket `regex` repräsentiert. Wie alle Pakete, die Formeln repräsentieren, ist auch das Paket `regex` so organisiert, dass eine gemeinsame abstrakte Oberklasse `Expression` existiert, von der alle Operatoren und Konstanten erben. Indirekt erbt die Klasse `Letter` ebenfalls von `Expression`. Ein Basisausdruck ist entweder true oder false, dargestellt

durch eine Instanz der Klasse `BooleanLetter` oder ein Element des Eingabealphabets, dargestellt durch die Klasse `StringLetter`. Letztere Klasse besitzt einen beliebigen String als Attribut `_1`. Dass dieser tatsächlich Element des Alphabets ist, kann an dieser Stelle nicht sichergestellt werden, da ein einzelner Ausdruck einer Formel nicht über das Alphabet verfügt. Entsprechend muss diese Konsistenzprüfung von außen bei der Verwendung dieser Klasse durchgeführt werden. Dazu existiert in Automatenklassen die Methode `checkAlphabet`, auf die in Abschnitt 5.4 auf Seite 85 näher eingegangen wird.

5.3.1. Textuelle Repräsentation

In Kapitel 2 wurde zu jeder verwendeten Formel eine Grammatik in EBNF angegeben. Aus dieser Grammatik ergibt sich unmittelbar die textuelle Repräsentation der Formeln. Die verwendeten mathematischen Symbole werden dabei entsprechend Tabelle 5.1 auf Seite 83 mithilfe von ASCII-Zeichen dargestellt. Dabei können viele Zeichen sowohl durch Klammern oder andere Sonderzeichen als auch durch Buchstaben dargestellt werden. Beide Notationen sind im Folgenden gleichwertig und können beliebig ersetzt werden. Grundsätzlich gilt für die textuelle Repräsentation von Formeln und Automaten in dieser Arbeit: Basisausdrücke, also Elemente des Eingabealphabets, werden mit Kleinbuchstaben, Operatoren und Konstanten hingegen mit Großbuchstaben bezeichnet. Durch diese Unterscheidung können alle Buchstaben auch zur Kennzeichnung von Operatoren verwendet werden.

In Tabelle 5.1 auf Seite 83 werden alle mathematischen Symbole der in dieser Arbeit verwendeten Formeln aufgeführt. Die Operatoren sind nur innerhalb einer Sorte Formeln eindeutig, deswegen werden wir im Folgenden eine Kennzeichnung der Formelart in der textuellen Repräsentation kennen lernen. Weiter fällt auf, dass der Vergangenheits-Operator $-x$ auf Basisausdrücken x und der allgemeine Vergangenheits-Operator x^{-1} auf regulären Ausdrücken x beide durch `-x` dargestellt werden. Dies ist möglich, da beim Parsen einer Formel der Art `-x` der Operator zunächst immer als allgemeiner Vergangenheits-Operator interpretiert wird. In der weiteren Verarbeitung wird der allgemeine Vergangenheits-Operator dann durch den Vergangenheits-Operator auf Basisausdrücken ersetzt. Letzterer wird also eigentlich nur in Zwischenzuständen benötigt.

Ein Basisausdruck wird immer entweder als Folge von Ziffern und Kleinbuchstaben oder als beliebige Zeichenkette in Zollzeichen (`"`) angegeben. Eine Folge von Ziffern und Kleinbuchstaben entspricht dabei der gleichen Zeichenfolge innerhalb von Zollzeichen. Auch bei den beliebigen Zeichenketten wird streng zwischen Groß- und Kleinschreibung unterschieden. Damit bei der textuellen Repräsentation immer die Art der verwendeten Formel bekannt ist und ein Alphabet mit angegeben werden kann, wird die folgende Grammatik in EBNF verwendet, um Formeln zu parsen.

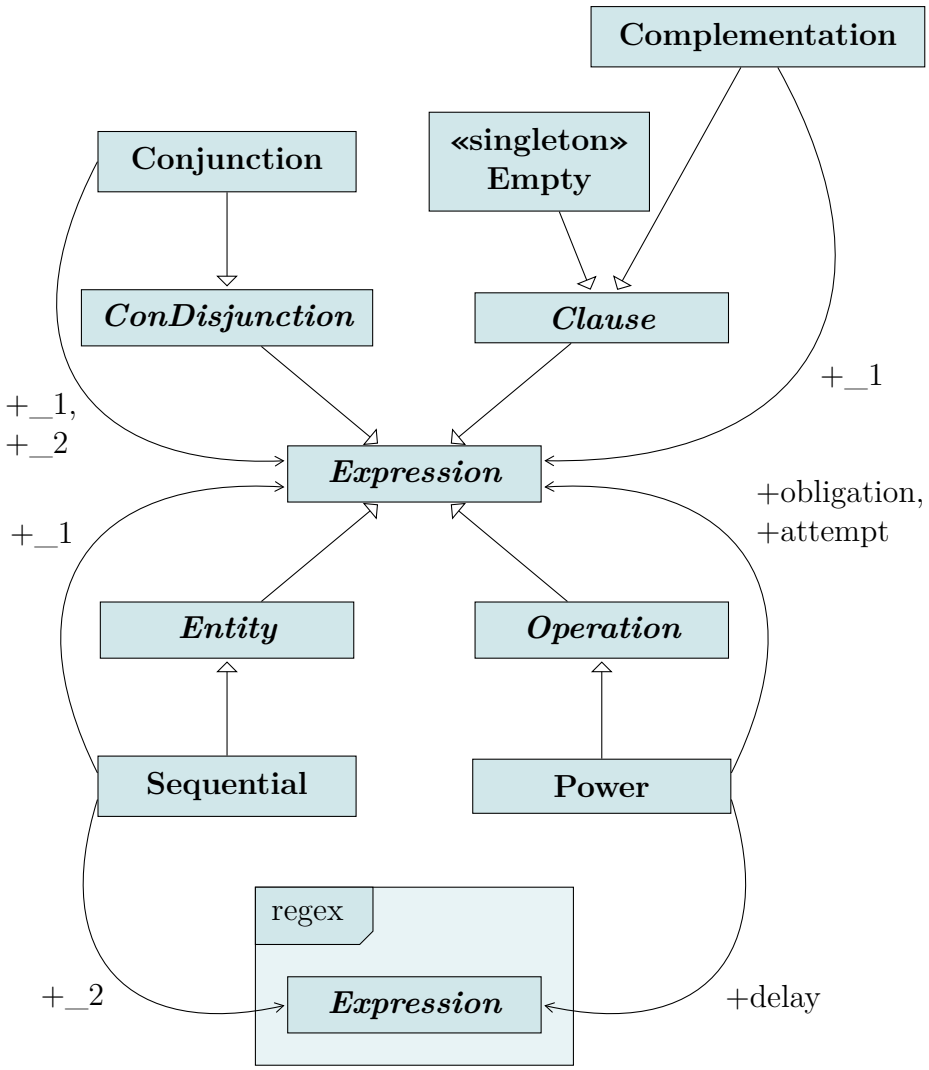

Abbildung 5.3.: UML-Klassendiagramm der Struktur der Klassen, die den Aufbau einer RLTL-Formel speichern. In der Abbildung steht die Klasse `Conjunction` stellvertretend für die Klasse `Conjunction` und die Klasse `Disjunction`, die Klasse `Sequential` stellvertretend für die Klasse `Sequential` und die Klasse `UniSequential` und die Klasse `Power` stellvertretend für die Klasse `Power`, die Klasse `WeakPower`, die Klasse `UniPower` und die Klasse `UniWeakPower`.

Logik	Name	Symbol	Klammern	Buchstaben
RE	Disjunktion	$x\|y$	`x	y`	
RE	Konkatenation	xy	`x y`	
RE	Kleene-Operator	$x*y$	`x * y`	
RE	Past	x^{-1}	`-x`	
RE	Basic-Past	$-p$	`-p`	
ORE	Disjunktion	$x\|\|y$		`x OR y`
ORE	Pumpoperator	$x;y^{\omega}$	`x # y`	
RLTL, LTL	Konjunktion	$a \wedge b$	`a && b`	`a AND b`
RLTL, LTL	Disjunktion	$a \vee b$	`a || b`	`a OR b`
LTL	Implikation	$a \rightarrow b$	`a -> b`	`a IMP b`
RLTL	Power	$a/x\rangle\rangle b$	`a / x >> b`	
RLTL	schw. Power	$a/x\rangle b$	`a / x > b`	
RLTL	duale Power	$a//x\rangle\rangle b$	`a // x >> b`	
RLTL	duale schw. Power	$a//x\rangle b$	`a // x > b`	
RLTL	Konkatenation	$x;a$	`x ; a`	
RLTL	duale Konkatenation	$x;;a$	`x ;; a`	
RLTL	leere Sprache	$\emptyset$	`%`	`EMPTY`
RLTL, LTL	Negation	$\neg a$	`!a`	`NOT a`
LTL	Until	$a\,\mathcal{U}\,b$		`a U b`
LTL	Back	$a\,\mathcal{B}\,b$		`a B b`
LTL	Weak-Unitl	$a\,\mathcal{W}\,b$		`a W b`
LTL	Release	$a\,\mathcal{R}\,b$		`a R b`
LTL	Since	$a\,\mathcal{S}\,b$		`a S b`
LTL	Next	$\bigcirc a$	`() a`	`X a`
LTL	Previous	$\ominus a$	`(*) a`	`P a`
LTL	Weak-Previous	$\circledcirc a$	`(+) a`	`Q a`
LTL	Finally	$\Diamond a$	`<> a`	`F a`
LTL	Globally	$\Box a$	`[] a`	`G a`
LTL	Past-Finally	$\diamondminus a$	`<*> a`	`O a`
LTL	Past-Globally	$\boxminus a$	`[*] a`	`H a`

Tabelle 5.1.: Darstellung der in den Formeln verwendeten mathematischen Symbole mit dem ASCII-Zeichensatz

Die Angabe eines Alphabets ist dabei optional. Wird kein Alphabet angegeben, so wird aus den verwendeten Basisausdrücken ein Alphabet erraten.

```
FORMULA = E | ["," A]
      E = "RLTL" "=" RLTL              // Formelausdruck
        | "REGEX" "=" RE
        | "LTL" "=" LTL
        | "OMEGAREGEX" "=" ORE
      A = "ALPHABET" "=" "[" C "]"     // optionale Alphabetsangabe
      C = {S ","} S                    // Folge von Zeichen
      S = P | "EPSILON"                // Zeichen des Alphabets
```

Dabei ist `P` wie oben beschrieben eine Zeichenkette in Zollzeichen oder eine Folge von Ziffern und Kleinbuchstaben.

5.3.2. Ausgabe in der textuellen Repräsentation

Um Formeln, die in der hier beschriebenen Datenstruktur vorliegen, in die textuelle Repräsentation umzuwandeln, wandelt jeder Operator sich selbst in einen String um und setzt anstelle der Operanden die textuellen Repräsentationen dieser ein. Für eine möglichst elegante Verwendung von Formeln in Scala wird diese Übersetzung in eine textuelle Repräsentation in der `toString`-Methode realisiert. Diese ruft nacheinander die `toString`-Methode aller Operanden auf, die als Attribute gespeichert sind, und setzt diese zu einem eigenen Rückgabewert zusammen. Diese Rekursion bricht bei Konstanten und Basisausdrücken ab und liefert eine flache Form des Baumes, den die Datenstruktur der Formel bildet. Damit diese Struktur des Baumes erhalten bleibt, müssten alle Operanden bei der Ausgabe jeweils geklammert werden. Auf diese Weise entstehen aber viele überflüssige Klammern, da die Rangfolge der Operatoren eine implizite Klammerung vorgibt. Um diese bei der Ausgabe zu beachten, besitzt jede abstrakte Klasse, die eine Ebene in der Rangfolge der Operatoren darstellt, eine Methode mit der Signatur

```
def brackets(e: Expression): String
```

Diese Methode liefert die textuelle Repräsentation eines Ausdrucks und klammert diesen genau dann, wenn der oberste Operator dieses Ausdrucks auf gleicher oder höherer Ebene in der Rangfolge der Operatoren liegt.

5.4. Parser

Bei der primären Verwendung des im Rahmen dieser Arbeit implementierten Programms wird eine RLTL-Formel eingelesen und in einen Automaten umgewandelt. Alternativ kann auch eine LTL-Formel eingelesen und zunächst in eine RLTL-Formel umgewandelt werden. Dabei wird die Formel in der bereits beschriebenen textuellen Repräsentation eingegeben und muss zunächst mit Hilfe eines Parsers interpretiert und in die entsprechende Datenstruktur umgewandelt werden, die dann weiter verarbeitet werden kann. Um einen effizienten Parser zu erzeugen stehen für die meisten Programmiersprachen Parsergeneratoren wie Bison oder Yacc zur Verfügung. Diese liefern meist sehr gute Ergebnisse und könnten auch für diese Aufgabe eingesetzt werden. Sie sind aber relativ kompliziert zu verwenden und benötigen jeweils einen zusätzlichen Kompilierungsschritt, da die Eingabe für den Parsergenerator zunächst in zum Beispiel Java-Quellcode übersetzt wird und dieser anschließend kompiliert wird. Im Gegensatz dazu bietet Scala mit funktionaler Programmierung und Parserkombinatoren eine sehr elegante und flexible Möglichkeit, Parser direkt in Scala zu spezifizieren und als normale Methoden direkt zu verwenden (vergleiche [OSV08, Kapitel 31]).

Die grundsätzliche Idee von Parserkombinatoren besteht darin, einfache Parser zu komplexeren Parsern zu kombinieren. Existiert zum Beispiel ein Parser für das Schlüsselwort `RLTL` und ein Parser für das Schlüsselwort `LTL`, so können diese beiden Parser jeweils nur genau ein Wort erkennen. Werden diese nun derart zu einer Disjunktion kombiniert, dass zunächst der erste Parser versucht, sein Wort zu erkennen, und im Fehlerfall der zweite Parser versucht, sein Wort zu erkennen, so entsteht ein Parser, der beide Worte erkennen kann. Dabei werden die Parser jeweils mit der aktuellen Position in der Eingabe aufgerufen, sodass durch die Rekursion ein Backtracking entsteht. Die zweite elementare Kombination von Parsern ist die sequenzielle Kombination, bei der beide Parser nacheinander akzeptieren müssen, wobei der zweite Parser an der Stelle in der Eingabe beginnt, wo der erste Parser endet. Stehen diese beiden Parserkombinatoren zur Verfügung, kann ein LL($*$)-Parser generiert werden. Ein LL(k)-Parser ist dabei ein Parser, der die Eingabe von links nach rechts abarbeitet und dabei jeweils versucht die Nonterminalsymbole der Grammatik entsprechend den Ableitungsregeln zu ersetzen. Dabei blickt der Parser höchstens k Schritte voraus, um sich für die richtige Regel zu entscheiden. Parserkombinatoren realisieren LL($*$)-Parser, da durch das Backtracking theoretisch beliebig weit voraus geschaut werden kann. Wir werden aber sehen, dass in der Praxis an diesem Punkt ineffiziente Parser entstehen.

Ein LL($*$)-Parser versucht auch eine fehlerhafte Eingabe solange zu interpretieren, bis sich kein passender Parser mehr findet. Aus diesem Grunde werden Fehlermeldungen häufig deutlich nach dem eigentlichen Fehler in der Eingabe generiert. Hier

sind manuell geschriebene Parser die bessere Alternative, weil diese auf typische Fehler angepasst werden können. Da das Ziel der Implementierung aber unter anderem auch eine gute Nachvollziehbarkeit und Wiederverwendbarkeit des entstandenen Quellcodes ist, werden wir die Parserkombinatoren aus der Scala-API verwenden. Die zur Verfügung stehenden Kombinatoren entsprechen gerade den syntaktischen Elementen der EBNF, sodass die in Kapitel 2 eingeführten Grammatiken in EBNF mit einigen syntaktischen Änderungen direkt als Parserkombinatoren in Scala verwendet werden können. Für die genaue Funktionsweise der im Folgenden verwendeten Parserkombinatoren aus der Scala-API sei auf die Erläuterungen von Daniel Spiewak in [Spi11] verwiesen.

In der Klasse `FormulaParserCombinators` wird die oben angegebene Grammatik `FORMULA` durch die öffentliche Methode `formula` realisiert. Wir wollen beispielhaft die Methoden dieser Klasse genauer betrachten.

```
def formula = expression ~ opt("," ~> alphabet) ^^ {
  case expression ~ Some(alphabet) =>
    Formula(expression, Set(alphabet:_*)).checkAlphabet
  case expression ~ None =>
    Formula(expression)
}

private def expression: Parser[Expression] =
  "RLTL" ~> "=" ~> rltl |
  "REGEX" ~> "=" ~> regex |
  "LTL" ~> "=" ~> ltl |
  "OMEGAREGEX" ~> "=" ~> omegaregex
```

Aus der Ableitungsregel

```
FORMULA = E | ["," A]
```

wurde folgende Methodendefinition:

```
def formula = expression ~ opt("," ~> alphabet)
```

Der Operator ~ übernimmt dabei die Konkatenation von zwei Parsern zu einem neuen Parser und die Methode `opt` realisieren ein optionales Element. Wird dem Operator ~ noch ein > nachgestellt, so wird nur der zweite Operand in das Ergebnis des Parsers übernommen. In diesem Fall hat das Komma `","` für die weitere Verarbeitung keine Bedeutung. Es wird nur für die eindeutige Trennung von Formel und Alphabet benötigt. Ein Parsergenerator liefert das Ergebnis in einer Datenstruktur, in der die Tokens entsprechend der verwendeten Parserstruktur angeordnet sind. Prinzipiell kann diese Datenstruktur nach dem Parsen ausgewertet

und in die passende Datenstruktur der Formel umgewandelt werden. Die einzelnen Elemente können aber auch schon während des Parsens entsprechend ausgewertet werden. Dafür wird der Operator `^^` verwendet. Nach diesem Operator wird eine Methode angegeben, die die geparste Struktur weiter verarbeitet. Wenn die verwendeten Parser ebenfalls bereits eine Auswertung vorgenommen haben, muss auch die Auswertung im Kombinator die Ergebnisse nur noch kombinieren. Bei der Methode `formula` wird in dieser Methode mit Pattern-Matching geprüft, ob ein Alphabet angegeben wurde oder nicht. Wurde explizit mit der Formel ein Alphabet angegeben, so wird ein Objekt der Klasse `Formula` mit der Formel und dem Alphabet erzeugt. Auf dieses Objekt wird nun die Methode `checkAlphabet` aufgerufen. Da die einzelnen Ausdrücke der Formel nicht über das Alphabet verfügen, muss nach dem Zusammenführen eines Ausdrucks mit einem Alphabet sichergestellt werden, dass in der Formel nur Elemente des Alphabets als Basisausdrücke verwendet werden. Für diese Prüfung stellen alle abstrakten Klassen `Expression` jeweils eine Methode `checkAlphabet` zur Verfügung, der ein Alphabet übergeben werden muss. Alle erbenden Realisierungen von Operatoren und Konstanten müssen die Methode implementieren. Operatoren rufen einfach wieder `checkAlphabet` auf alle Operanden auf und Basisausdrücke prüfen, ob sie Element des übergebenen Alphabets sind. Auf diese Weise steigt die Prüfung rekursiv in den Formelbaum ab. Wird ein Basisausdruck gefunden, der nicht Element des Alphabets ist, so wird eine Ausnahme `SignNotFoundException` geworfen. Diese muss vom Anwender der Bibliothek bzw. von der Kommandozeilenanwendung behandelt werden.

Wird kein Alphabet explizit angegeben, so wird die Datenstruktur der Formel nur aus dem Ausdruck generiert. In der `apply`-Methode des Kompagnonobjektes von `Formula` wird dann ein minimales Alphabet aus dem Ausdruck erzeugt. Die Methode `guessAlphabet` funktioniert ganz analog zur Überprüfung des Alphabets in `checkAlphabet`, nur dass hier jeder Basisausdruck sein Zeichen dem Alphabet hinzufügt. Die Methode gibt das jeweilige Alphabet zurück und Implementierungen von Operatoren kombinieren die Alphabete der Operanden zu einem gemeinsamen Alphabet. Auf diese Weise kann ein Alphabet generiert werden, dass alle verwendeten Zeichen der Formel enthält. Mit einem auf diese Weise generierten Alphabet können zwar alle Umwandlungsschritte durchgeführt werden, da alle verwendeten Zeichen der Formel auch Zeichen des Alphabets sind, aber mit diesem minimalen Alphabet werden nicht immer sinnvolle Ergebnisse erzielt. Betrachtet man zum Beispiel die Implikation $a \rightarrow b$, so kann diese auch als $\neg a \vee b$ geschrieben werden. Existieren nur die Zeichen a und b im Alphabet, so kann der Ausdruck zu $b \vee b$ und damit zu b vereinfacht werden. Damit der Sinn einer Implikation bei der Umwandlung in einen Automaten nicht entstellt wird, muss das Alphabet daher mindestens noch ein weiteres nicht beteiligtes Zeichen enthalten.

In der Methode `expression` werden die Methoden der verschiedenen Formelarten verwendet. Für jede Formel existiert ein eigener Trait mit den entsprechenden Par-

serkombinatoren. In Abbildung 5.4 auf Seite 91 wird die Beziehung zwischen den Traits und Klassen, die am Parsen von Formeln beteiligt sind, dargestellt. Die Parserkombinatoren der einzelnen Formelarten sind jeweils in einem Trait zusammengefasst, damit die Klasse `FormulaParserCombinators` alle Traits einmischen kann. Alle vier Traits erben von `JavaTokenParsers`, wo grundlegende Parser für Zeichenketten und Ähnliches über reguläre Ausdrücke realisiert werden. Für komplexe Parser sind reguläre Ausdrücke zwar sehr ineffizient (und unübersichtlich), aber zum Beispiel die Suche nach Folgen von Kleinbuchstaben und Ziffern kann damit sehr elegant notiert werden. Der Trait `AlphabetParserCombinators` stellt die öffentliche Methode `alphabet` zur Verfügung. Da Alphabete von Automaten und Formeln gleichermaßen benötigt werden, wird hier derselbe Parser verwendet. Soll einer der Parser in den Traits zu Testzwecken alleine verwendet werden, kann der Trait in eine anonyme Klasse eingemischt werden. So wird zum Beispiel in den Unit Tests für den LTL-Parser indirekt eine Instanz des Traits `LtlParserCombinators` erzeugt und damit eine Testeingabe `s` verarbeitet.

```
val parser = new LtlParserCombinators{}
parser.parseAll(parser.ltl, s) match {
  case parser.Success(parsed, _) => {
    assertEquals("formula has to be ...", given, parsed)
  }
  case x => fail(x.toString)
}
```

Man beachte die Klammern `{}` am Ende des Traitnamens. Diese entsprechen einer anonymen Klasse ohne Methoden, die den Trait einmischt. Die Methode `parseAll` erhält einen Parser und eine Zeichenkette als Parameter und liefert entweder eine Instanz der Klasse `Success` oder eine Instanz einer Fehlerklasse zurück.

Zum Abschluss des Abschnitts über Formelparser wollen wir noch ein Problem näher betrachten, dass insbesondere bei LTL-Formeln auftritt. In der Operatorrangfolge von LTL befinden sich sehr viele Operatoren auf einer Ebene. So haben zum Beispiel alle temporalen binären LTL-Operatoren (Until, Weak-Until, Release, Back und Since) den gleichen Rang. Der naive Ansatz besteht nun darin, den Parsergenerator direkt entsprechend folgender Ableitungsregel aus der LTL-Grammatik zu realisieren.

`B = U "`$\mathcal{U}$`" B | U "`$\mathcal{W}$`" B | U "`$\mathcal{R}$`" B | U "`$\mathcal{B}$`" B | U "`$\mathcal{S}$`" B | U`

Ist der nächste zu parsende Ausdruck nun aber kein binärer Operator, so muss die Ableitung von `B` nach `U` verwendet werden. In diesem Fall wird zunächst für jeden binären Operator eine Ableitung nach `U` versucht, um dann festzustellen, dass der Operator im Anschluss an die Auswertung von `U` nicht vorkommt. Ein ganz analoges Problem ergibt sich für die unären Operatoren. Besonders kritisch wird diese

Vielzahl an Möglichkeiten, die jeweils durchprobiert werden müssen, bei stark geklammerten Ausdrücken. Denn die Klammerung muss jeweils als letzte Ableitungsregel angegeben werden, damit Klammern in der Rangfolge der Operatoren nicht stärker binden als andere Operatoren. Aus diesem Grunde ist die Laufzeit dieses naiven Ansatzes schon beim Parsen eines mehrfach geklammerten Basisausdrucks wie `(((((((((c)))))))))` nicht mehr annehmbar, denn jede Möglichkeit, die Ableitungsregeln anzuwenden, muss bis zum Ende durchprobiert werden, damit sie ausgeschlossen werden kann. Eine deutlich effizientere Umsetzung kombiniert alle Operatoren einer Ebene in der Rangfolge zu einer einzigen Ableitungsregel.

```
B = U ("𝒰" | "𝒲" | "ℛ" | "ℬ" | "𝒮") B | U
```

So entsteht für alle Operatoren gleichen Ranges nur *eine* neue Möglichkeit eine öffnende Klammer zu behandeln. Um dieses Verfahren elegant implementieren zu können, besitzen die Kompagnonobjekte der Operator-Klassen jeweils ein Attribut `charSymbol` und `braceSymbol`. Damit kann aus einem solchen Objekt in folgender Methode ein Parser für genau diesen Operator erzeugt werden. Da nicht für jeden Operator zwei textuelle Repräsentationen zur Verfügung stehen, wird dabei zunächst geprüft, ob neben dem `charSymbol` auch `braceSymbol` angegeben wurde.

```
private def getOperatorParser(operator: Operator):
  Parser[Operator] = {
    operator.braceSymbol match {
      case Some(op) =>
        (operator.charSymbol | op) ^^
          { case _ => operator }
      case None =>
        operator.charSymbol ^^
          { case _ => operator }
    }
}
```

Dieser Parser gibt direkt das zum Operator passende Kompagnonobjekt zurück, dessen `apply`-Methode im Folgenden angewendet werden kann, sodass die richtige Datenstruktur entsteht, ohne das weiter zwischen den verschiedenen Operatoren unterschieden werden muss. Mit der Methode `createOperatorDisjunction` kann nun wiederum aus einer Liste von Operatoren ein Parser generiert werden, der alle diese Operatoren akzeptiert.

```
private def createOperatorDisjunction
  (list: List[Operator]): Parser[Operator] =
    list.map { op => getOperatorParser(op) } match {
      case head :: list => list.foldLeft(head) {
```

```
      case (base, op) => base | op
    }
  }
private val binaryOperators =
  List(Until, WeakUntil, BackTo, Release, Since)
private val binaryOperatorParsers =
  createOperatorDisjunction(binaryOperators)
```

Aus oben angegebener Ableitungsregel wird auf diese Weise die Methode

```
private def binary: Parser[Expression] =
  unary ~ binaryOperatorParsers ~ binary ^^ {
    case _1 ~ (operator: BinaryOperator) ~ _2 =>
      operator(_1, _2)
  } | unary
```

Hier wird implizit die `apply`-Methode des entsprechenden Kompagnonobjektes des gefundenen Operators auf die beiden Operanden angewendet.

5.5. Automaten

Wie wir bereits gesehen haben, erben alle Automaten von der abstrakten Klasse `Automata`. Anders als bei Formeln speichert jeder Automat sein eigenes Alphabet, da ein Automat jeweils als eine Instanz der Klasse `Nfa` bzw. der Klasse `Apw` repräsentiert wird. Wie bereits in der Einleitung erläutert, werden Automaten sehr dicht an der mathematischen Notation gespeichert. So besteht ein NFA aus

- einem Alphabet,
- einer Liste von Zuständen,
- einer Liste von Startzuständen,
- einer Tabelle, die die Transitionsrelation als Zuordnung von Zuständen und Zeichen des Alphabets auf eine Liste von Zuständen realisiert
- und einer Liste von akzeptierenden Zuständen.

Ein Zustand ist dabei eine Instanz der Klasse `State`. Instanzen dieser Klasse werden zunächst nicht aufgrund des Namens des Zustandes, sondern direkt über die Objektidentität unterschieden. Auf diese Weise erhält ein neuer Zustand keinen Namen und beim Kombinieren von mehreren Automaten zu einem neuen Automaten muss nicht auf die Konsistenz der Zustandsbezeichnungen geachtet werden. Wichtig ist nur, dass für verschiedene Zustände jeweils eine neue Instanz der Klasse `State` erzeugt wird. Erst vor der Ausgabe eines Automaten in seiner textuellen Repräsentation erhält jeder Zustand einen eindeutigen Namen.

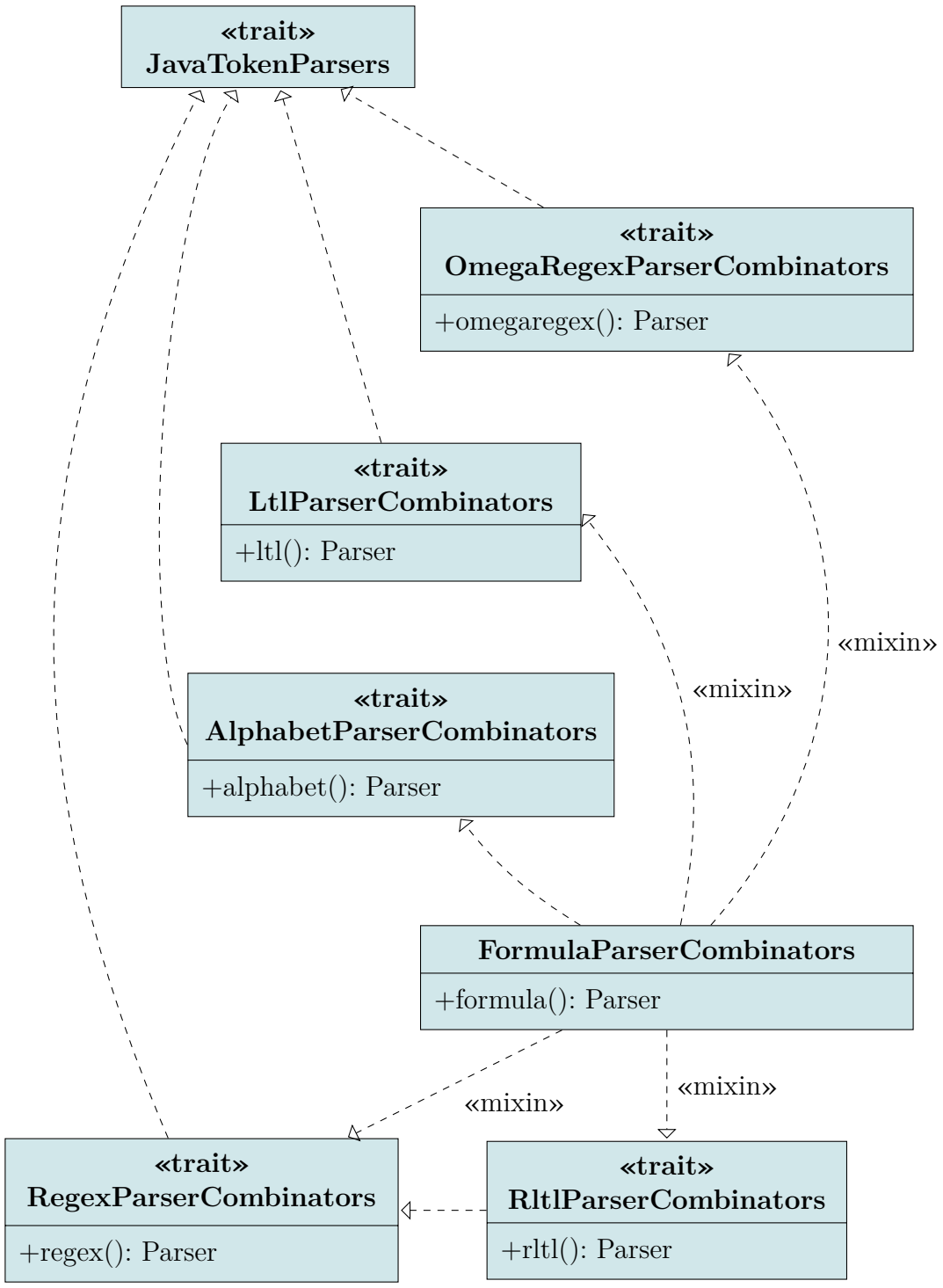

Abbildung 5.4.: UML-Klassendiagramm aller am Parsen von Formeln beteiligten Klassen und Traits.

Die abstrakte Oberklasse `Automata` schreibt nur die Methoden `toString` und `toDot` vor, die für die Ausgabe eines Automaten im eigenen textuellen Austauschformat bzw. im speziellen Eingabeformat für externe Graphenvisualisierungssoftware zuständig sind. Entsprechend werden alle weiteren Methoden direkt in den Automaten definiert. Hier existieren zwar einige Namenskonventionen, aber ein allgemeiner Automat muss zunächst in einen APW oder NFA gecastet werden, bevor diese Methoden verwendet werden können. Dieses Prinzip wurde verwendet, weil die beiden Automatentypen sich in der Praxis sehr stark unterscheiden, sodass selten beide Automatentypen generisch gleich behandelt werden können. Verschiedene Algorithmen, wie zum Beispiel das Vergeben von eindeutigen Namen für alle Zustände der Zustandsmenge, werden durchaus in beiden Automatentypen benötigt, sodass ein gewisser Grad an Codewiederholung entsteht. Vor allem die Transitionsrelation bzw. -funktion in den beiden Automatentypen werden sehr unterschiedlich repräsentiert. Somit würde eine generalisierte Lösung für diese Algorithmen in der abstrakten Oberklasse die Übersichtlichkeit und Nachvollziehbarkeit des Quellcodes stark verschlechtern, da immer wieder zwischen den beiden Automatentypen unterschieden werden müsste oder viele wenig intuitive Hilfsmethoden in den Automaten eingeführt werden müssten. Gemeinsam von beiden Automatentypen verwendet werden nur die Klasse `State`, die Klasse `Sign`, die ein Zeichen der Eingabe repräsentiert und auch von den Formeln und dem Alphabet verwendet wird, und die Klasse `Direction`, die eine Bewegungsrichtung eines Zwei-Wege-Automaten speichert. Die Klasse `DirectedState` speichert einen Zustand und eine Bewegungsrichtung und wird verwendet, um die Zielelemente einer Transition im Zwei-Wege-Automaten zu speichern. Weiter wird der Operator `::` definiert, um elegant einen Zustand und eine Bewegungsrichtung zu einem mit der Bewegungsrichtung annotierten Zustand kombinieren zu können. So erzeugt

```
val ds = State() :: Forward
```

einen neuen Zustand, der mit der Bewegungsrichtung +1 kombiniert wird. Damit in der Datenstruktur nicht zwischen Zwei-Wege- und Ein-Wege-Automaten, sowie Automaten mit ε-Transitionen unterschieden werden muss, werden alle Automaten intern als Zwei-Wege-Automaten mit ε-Transitionen behandelt. Erst bei der Ausgabe eines Automaten in textueller Repräsentation wird überprüft, ob der Automat ε-Transitionen bzw. Transitionen mit einer anderen Bewegungsrichtung als +1 enthält. Entsprechend wird dann ein Automat ausgegeben, der möglichst wenig benötigt, sodass dieser möglichst einfach weiter verarbeitet werden kann.

Die Transitionsfunktion von NFAs kann als Tabelle gespeichert werden. Als Schlüssel wird dabei ein Zustand und ein Eingabezeichen verwendet und als Wert eine Liste über der Klasse `DirectedState`, die einen Zustand und eine Bewegungsrichtung speichert. Kann beim NFA für die Menge der möglichen Folgezustände

einfach eine Liste verwendet werden, so müssen beim APW an dieser Stelle positive boolesche Formeln über der Klasse `DirectedState` gespeichert werden. Da neben einer solchen positiven booleschen Formel unter anderem für die initialen Zustände auch Formeln über Zuständen ohne Bewegungsrichtung und für die Entfernung von ε-Transitionen darüber hinaus Formeln über Paare von Zuständen und Akzeptanzbedingungen benötigt werden, wurde für positive boolesche Formeln eine generische Datenstruktur erzeugt. Diese kann dann je nach Bedarf mit einer anderen Klasse konkretisiert werden. Abbildung 5.5 auf der nächsten Seite zeigt alle an dieser Datenstruktur beteiligten Klassen. Der zu speichernde Datentyp wird von einer Klasse `BoolElement` gekapselt. Zusammen mit den Konstanten `BoolFalse` und `BoolTrue` erben diese von der abstrakten Klasse `BoolPiece`. Die beiden Konstanten sind allerdings Singleton-Objekte, da die verschiedenen Vorkommen von true bzw. false in einer Formel nicht unterschieden werden müssen. `BoolPiece` erbt wiederum zusammen mit den Operatoren `BoolAnd` und `BoolOr` von der abstrakten Oberklasse `PosBool`. Die beiden Operatoren besitzen jeweils zwei Operanden vom Typ `PosBool`, sodass auf diese Weise die Baumstruktur der positiven booleschen Formel repräsentiert werden kann.

Die Generalisierung der Klasse `PosBool` und der erbenden Klassen findet nicht nur mit einem Platzhalter `A` für eine beliebige Klasse, sondern mit `+A` statt. Auf diese Weise erbt eine Konkretisierung `PosBool[A]` von einer anderen Konkretisierung `PosBool[B]`, wenn die Klasse `A` von der Klasse `B` erbt. Diese Form der Generalisierng wird auch bei vielen Scala-Klassen wie zum Beispiel der Klasse `List` verwendet, wird aber hier insbesondere benötigt, damit die beiden Singleton-Objekte der Konstanten von `PosBool[Nothing]` erben können, denn als Objekte können diese nicht in einer generalisierten Form existieren. Die Klasse `Nothing` ist dabei eine spezielle Scala-Klasse, die von allen anderen Klassen erbt, von der aber keine Instanzen erzeugt werden können.

Zu den beiden Operator-Klassen existieren die Kompagnonobjekte `BoolAnd` und `BoolOr`, die beide von der gemeinsamen abstrakten Oberklasse `BoolOperator` erben. Diese schreibt ein Attribut `other` vom Typ `BoolOperator` vor, dass jeweils auf den anderen Operator zeigt. Auf diese Weise können die Kompagnonobjekte sehr elegant verwendet werden, um positive boolesche Formeln zu generieren. Um einfach aus Listen von Objekten eine positive boolesche Formel dieser Objekte erstellen zu können, stellt die Klasse `BoolOperator` verschiedene weitere `apply`-Methoden zur Verfügung, die auf die `apply`-Methode mit zwei Parametern reduziert wird. Die Klasse `PosBool` stellt schließlich noch die aus dem `Collection`-Trait aus der Scala-API bekannten Methoden `map`, `flatMap` und `foreach` zur Verfügung. Die Klasse `PosBool` kann aber das Trait `Collection` nicht direkt einbinden, da eine positive boolesche Formel in der Struktur eher einem Baum als einer bloßen Sammlung von Objekten entspricht. Daher bietet die Klasse `PosBool` die Methode `toList` an, die alle Objekte der Formel in einer Liste zurückgibt. So kann zum Bei-

spiel sehr einfach die `contains`-Methode realisiert werden, in dem `contains` auf der Liste aufgerufen wird. Auf diese Weise kann aber unter anderem auch überprüft werden, ob ein Zustand `q` in einer Formel `f` vom Typ `PosBool[DirectedState]` vorhanden ist. Dazu genügt folgender Aufruf:

```
f.toList.map {
  case DirectedState(direction, state) => state
}.contains(q)
```

Dabei werden alle mit Bewegungsrichtungen annotierten Zustände aus der Formeln in einer Liste gesammelt und diese Liste dann in eine neue Liste umgewandelt, in der sich nur die Zustände ohne die Bewegungsrichtungen befinden. Auf diese Liste kann nun `contains` mit dem zu prüfenden Zustand `q` aufgerufen werden.

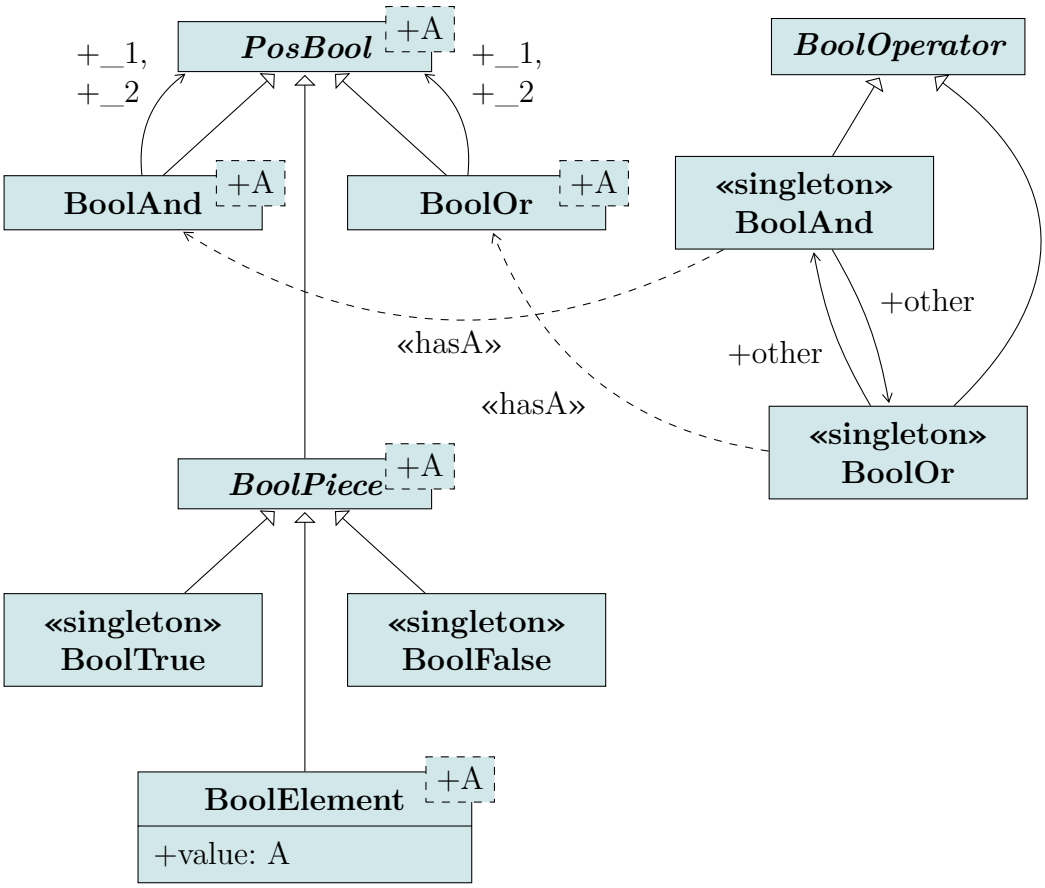

Abbildung 5.5.: UML-Klassendiagramm der zur Erzeugung positiver boolescher Formeln benötigten Klassen und Kompagnonobjekte.

5.5.1. Textuelle Repräsentation

Um Automaten in textueller Form darzustellen, bieten sich verschiedene Lösungen an. Das Eingabeformat für die freie Graphenvisualisierungssoftware Graphviz[1] als Austauschformat zu verwenden, hätte den Vorteil, dass ein Automat in textueller Repräsentation auch immer direkt ein Graph ist, der mit dem Graphviz-Tool dot als Grafik dargestellt werden kann. Diese ursprüngliche Idee, dieses Dot-Format als Austauschformat zu verwenden, wurde aber im Laufe der Realisierung wieder verworfen. Stattdessen wurde im Rahmen dieser Arbeit das Automatendateiformat (AFF) zur Speicherung von Automaten entworfen und das Dot-Format nur zur grafischen Ausgabe von Automaten und nicht als Austauschformat benutzt. Das Problem bei der Verwendung des Dot-Formats als Austauschformat für Automaten besteht darin, dass entweder rein visuellen Elementen eine Semantik gegeben wird, sodass zum Beispiel alle doppelt umkreisten Zuständen als akzeptierende Zustände verstanden werden, oder es wird nur eine Untermenge des Dot-Formats verwendet. Im ersten Fall ergibt sich die Schwierigkeit, dass es sehr viele verschiedene Möglichkeiten gibt, in Dot einen Zustand als doppelt umkreisten Zustand darzustellen und ein Parser alle diese Möglichkeiten erkennen und entsprechend interpretieren müsste. Im zweiten Fall ergibt sich das Problem, dass Änderungen, die das durch dot generierte Bild nicht verändern, dazu führen können, dass der Automat nicht mehr oder anders eingelesen wird. Beide Schwierigkeiten resultieren daraus, dass das Dot-Format nicht dafür gedacht ist, Semantiken zu den Graphen zu speichern, sondern nur das Aussehen eines Graphen.

Im Folgenden werden die Definitionen der textuellen Repräsentation der verschiedenen Automaten in AFF gegeben. Im Gegensatz zu den Formeln wird das Format von AFF nur als informeller Text und nicht als EBNF angegeben, da dieses Format sehr einfach gehalten ist und im Wesentlichen nur aus vielen gleichen Elementen besteht. Eine tiefe Verschachtelung oder eine komplexe Operator-Rangfolge wie bei den Formeln existiert nicht. Entsprechend einfach kann mit den Parserkombinatoren ein Parser für dieses Format erzeugt werden.

Wie bei den Formeln gilt auch bei den Automaten der Grundsatz, dass alle Konstanten und Methoden in Großbuchstaben geschrieben werden, während alle Basisausdrücke (also Elemente des Eingabealphabets) in Kleinbuchstaben notiert werden. Die Ausdrücke true und false werden dabei allerdings als `TRUE` und `FALSE` notiert. Strings in Zollzeichen werden hingegen unabhängig von Groß- und Kleinschreibung immer als ein Element des Alphabets interpretiert.

[1] siehe `http://www.graphviz.org/`

5.5.2. Textuelle Repräsentation von NFAs

AFF orientiert sich stark an der mathematischen Notation eines NFAs als Tupel. Ein Automat beginnt dabei mit dem Schlüsselwort *NFA* und es folgt die Definition des Automaten in geschweiften Klammern.

Eine Menge wird dargestellt als durch Komma getrennte Liste umschlossen von eckigen Klammern (`[` und `]`). Das Alphabet wird eingeleitet durch den String *ALPHABET*, ein Gleichheitszeichen und eine Menge aller Elemente des Alphabets. Ein Element des Alphabets ist eine alphanumerische Zeichenfolge von Kleinbuchstaben, die keine Sonderzeichen (auch keine Unterstriche) enthält oder ein beliebiger String, der von Zollzeichen (`"`) umschlossen wird. Auf diese Weise werden auch die Mengen der Zustände (*STATES*) und die Menge der Anfangszustände (*START*) definiert. Ein Zustand ist dabei ebenfalls eine alphanumerische Zeichenfolge von Kleinbuchstaben, die keine Sonderzeichen (auch keine Unterstriche) enthält. Auf einen Zustand kann ein Doppelpunkt (`:`) und eine Annotation folgen. Auf diese Weise werden alle akzeptierenden Zustände mit der Annotation *ACCEPTING* gekennzeichnet.

Die Zustandsübergangsfunktion wird dargestellt durch mehrere Einträge folgender Form: Auf das Schlüsselwort *DELTA* folgen in Klammern erst der aktuelle Zustand und danach getrennt durch ein Komma (`,`) das gelesene Symbol des Alphabets. Dabei steht das Fragezeichen (`?`) als Platzhalter für ein beliebiges Zeichen des Alphabets. Nach einem Gleichheitszeichen (=) folgt die Menge der Folgezustände.

Diese Elemente müssen in der genannten Reihenfolge vorkommen.

In AFF kann ein NFA zum Beispiel so repräsentiert werden:

```
NFA {
  ALPHABET = ["a", "b", "c"]
  STATES = [q0, q1: ACCEPTING, q2: ACCEPTING, q3]
  START = [q1, q2]
  DELTA(q0, "a") = [q1, q2]
  DELTA(q0, ?) = []
  DELTA(q0, "c") = [q3]
  DELTA(q3, "a") = [q2]
}
```

Bei einem ε-NFA muss zusätzlich explizit das Schlüsselwort *EPSILON* in das Alphabet aufgenommen werden.

Ein 2NFA beginnt mit dem Schlüsselwort *2NFA*. Die Funktion *DELTA* bildet auf eine Menge von annotierten Zuständen mit den Annotationen *BACK*, *PAUSE* und

FORWARD ab. Jeder Zustand braucht eine Annotation, die die Bewegungsrichtung angibt. Eine Transition sieht dann zum Beispiel so aus:

```
DELTA(q3, "a") = [q3: BACK, q2: PAUSE]
```

5.5.3. Textuelle Repräsentation von APWs

Ein APW beginnt mit dem Schlüsselwort *APW*. Als Annotation der Zustände wird hier die Parität des Zustands als positive Ganzzahl (inklusive 0) angegeben. Im Gegensatz zum NFA muss jeder Zustand eine Annotation – also eine Parität – besitzen. Das Alphabet eines APWs wird nur als Menge der echten Eingabezeichen angegeben. Das bedeutet insbesondere, dass *TRUE* und *FALSE* nicht explizit angegeben werden dürfen.

Weiter bilden *START* und *DELTA* nicht mehr auf Mengen ab, sondern auf positive boolesche Funktionen. Diese bestehen aus Zuständen oder den Schlüsselwörtern *TRUE* oder *FALSE*, die durch die Schlüsselwörter *AND* oder *OR* zu booleschen Formeln zusammengesetzt werden können. Der Operator *AND* bindet dabei stärker als *OR*, aber es sind Klammern erlaubt, um die Auswertungsreihenfolge zu beeinflussen. Ein APW in dieser Notation sieht dann zum Beispiel so aus:

```
APW {
  ALPHABET = ["a", "b", "c"]
  STATES = [q0: 0, q1: 0, q2: 1, q3: 2]
  START = q1 AND q2
  DELTA(q0, "a") = (q1 AND q2) OR TRUE
  DELTA(q0, "b") = q2
  DELTA(q0, "c") = FALSE
  DELTA(q3, "b") = q2
}
```

Bei einem ε-APW muss analog zum ε-NFA muss zusätzlich explizit das Schlüsselwort *EPSILON* in das Alphabet aufgenommen werden, auch wenn es sich bei diesem nicht um ein Eingabezeichen handelt.

Ein 2APW beginnt mit dem Schlüsselwort *2APW*. Die Funktion *DELTA* bildet nicht mehr auf positive boolesche Funktionen von Zuständen sondern von annotierten Zuständen ab. Es werden die Annotationen *BACK*, *PAUSE* und *FORWARD* wie beim 2NFA verwendet.

Dabei ist zu beachten, dass jeder Zustand eine eigene Annotation erhält, die die Bewegungsrichtung angibt, und dass die Werte *TRUE* und *FALSE* nicht annotiert werden.

Hier sei ein Beispiel für solche Elemente der Transitionsfunktion gegeben:

```
DELTA(q3, "a") = q3: BACK OR q2: PAUSE AND q1: FORWARD
DELTA(q3, "b") = TRUE
DELTA(q3, "c") = FALSE AND q2: BACK
```

5.5.4. Ausgabe in der textuellen Repräsentation

Die Ausgabe von Automaten in textueller Repräsentation wird von beiden Automatentypen sehr ähnlich durchgeführt. Es wird jeweils die Methode `toString` überschrieben, die zunächst `toNamedNfa` bzw. `toNamedApw` aufruft, und auf dem Ergebnis dann `toStringHandler` aufruft. Diese Methode nutzt die große Flexibilität von Scala aus und verwendet als eine der wenigen Verfahren in diesem Programm veränderliche Variablen. In einer Schleife wird über alle Zustände iteriert und jedem Zustand ein eindeutiger Name zugewiesen. Diese Namen werden dabei generiert aus »`q`« und einer Zahl, die solange hochgezählt wird, bis ein freier Name erzeugt wurde. Da die Klasse `State` eine unveränderliche Klasse ist, wird für jeden Zustand, der einen neuen Namen erhalten soll, ein entsprechender neuer Zustand angelegt, der den alten Zustand ersetzen soll. In einer `HashMap` wird dabei zu jedem zu ersetzenden Zustand der neue Zustand gespeichert. Nach der Generierung der neuen Zustände können dann im ganzen Automaten alle Zustände durch die neuen Zustände ersetzt werden.

Da die Zustände in der Liste aller Zustände keine definierte Reihenfolge haben, entsteht auf diese Weise zwar eine eindeutige, aber eine zufällige Benennung der Zustände. Insbesondere, da bei der Konstruktion der Automaten `HashMaps` und `HashSets` eingesetzt werden, die keine Reihenfolge der Elemente garantieren, kann die Reihenfolge der Zustände auch bei gleicher Ausgangsformel variieren. Daher kann das Programm für die gleiche Eingabe unterschiedliche, isomorphe Ausgaben erzeugen, da die Zustände jeweils anders benannt werden.

Da sowohl AFF als auch die Datenstruktur von Automaten an der mathematischen Darstellung eines Automaten als 5-Tupel angelehnt ist, kann die Methode `toStringHandler` den von `toNamedNfa` bzw. `toNamedApw` generierten Automaten direkt ausgeben. Dieses textuelle Format ist zwar ein gutes Austauschformat und kann auch manuell sehr einfach generiert werden, aber in vielen Fällen ist es sehr angenehm, eine Darstellung eines Automaten als Graph zur Verfügung zu haben. Aus diesem Grunde generiert die Methode `toDot` für beide Automatenklassen eine Ausgabe im Dot-Format. Aus einer Datei `automata.dot` in diesem Format kann zum Beispiel mit folgendem Befehl eine Grafik `automata.dot.png` im PNG-Format generiert werden:

```
dot -O -Tpng automata.dot
```

Es entstehen Graphen, die mit einem Unterschied der in dieser Arbeit eingeführten Notation entsprechen: Die mathematischen Symbole true, false, ε, $\wedge$ und $\vee$ werden durch die Texte TRUE, FALSE, EPSILON, AND und OR dargestellt. Ein entsprechendes Beispiel wird in Abschnitt 5.7 auf Seite 105 näher betrachtet. Die Methode `toDot` ruft intern ebenfalls zunächst die Methode `toNamedNfa` bzw. `toNamedApw` auf, um auf dem generierten Automaten die Methode `toDotHandler` aufzurufen. Um einen APW in das Dot-Format zu exportieren wird im Gegensatz zu NFAs die zusätzliche Hilfsfunktion `exportTerm` benötigt. Diese wird rekursiv auf eine positive boolesche Formel angewendet und generiert für dessen Baumstruktur die benötigten Knoten und Kanten im Graphen. Die Methode wird auf die initiale Zustandsbelegung und jede Transition im Automaten angewendet, um den Automaten als Graph zu exportieren.

5.5.5. Parsen der textuellen Repräsentation

Für die textuelle Repräsentation von Automaten in AFF steht auch ein Parser zu Verfügung, der die generierten Ausgaben wieder importieren kann. Für die eigentliche Aufgabe des hier implementierten Programms, die Umwandlung einer RLTL-Formel in einen Automaten, wird dieser Parser zwar nicht benötigt. Die Möglichkeit, Automaten einlesen zu können, ist aber sehr hilfreich, um Optimierungen an existierenden Automaten zu testen, oder um generierte Automaten in das Dot-Format zu konvertieren, um diese als Graph darzustellen. Automaten im Dot-Format können hingegen nicht wieder eingelesen werden. Dieses Format dient nicht als Austauschformat.

Die Beziehungen aller am Parsen von Automaten beteiligten Klassen sind in Abbildung 5.6 auf der nächsten Seite zusammengefasst. Damit beim Parsen von Formeln und Automaten der gleiche Alphabetparser verwendet werden kann, wurde das Trait `AlphabetParserCombinators` aus der eigentlichen Parserklasse ausgelagert. Dem Parser liegt auch hier das Trait `JavaTokenParsers` zugrunde. Die öffentliche Schnittstelle der Klasse `AutomataParserCombinators` besteht im Wesentlichen aus folgender Methode.

```
def automata = nfa | twNfa | apw | twApw
```

Damit möglichst viel Code von Zwei-Wege- und Ein-Wege-Automaten gemeinsam verwendet werden kann, besitzen die Methoden `nfaContent` und `apwContent`, die von `nfa`, `twNfa`, `apw` und `twApw` aufgerufen werden, den impliziten Parameter `twoWay` vom Typ `Boolean`. Da der Parameter implizit ist, wird er automatisch an alle aufgerufenen Methoden weitergereicht, die ebenfalls einen impliziten Parameter `twoWay` besitzen. Auf diese Weise können große Teile des Parsers für Ein-Wege- und Zwei-Wege-Automaten verwendet werden und erst beim Parsen von Zuständen in

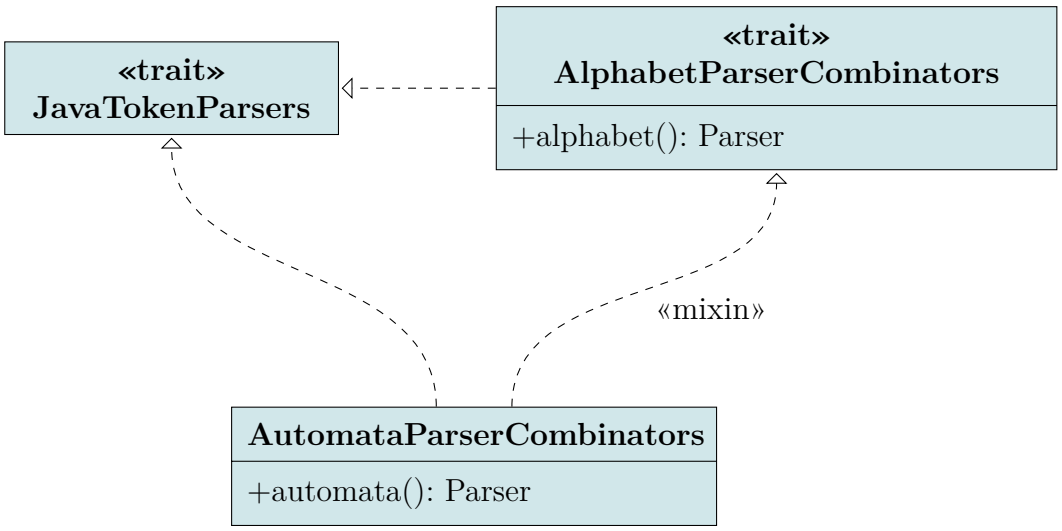

Abbildung 5.6.: UML-Klassendiagramm aller am Parsen von Automaten beteiligten Klassen und Traits.

Transitionsfunktionen müssen Zwei-Wege-Automaten speziell behandelt werden, sodass nach jedem Zustand eine Annotation zur Bewegungsrichtung erwartet wird. Ist ein Automat vollständig geparst, so wird von der Methode `nfaContent` bzw. der Methode `apwContent` die Methode `syncStatesByName` auf dem gerade generierten Automaten aufgerufen. Diese Methode löst das Problem, dass in Automaten Zustände nur dann als gleich angesehen werden, wenn sie das gleiche Objekt sind, während der Parser immer eine neue Instanz der Klasse `State` für jeden gelesenen Zustand generiert hat. Entsprechend werden alle in initialer Belegung, Transitionsrelationen und Akzeptanzbedingungen des Automaten vorkommenden Zustände durch gleichnamige Zustände aus der Zustandsmenge ersetzt. Wird dabei ein Zustand gefunden, der nicht in der Zustandsmenge vorkommt, so wird eine Ausnahme `SignNotFoundException` geworfen. Diese muss von Anwendern der API und der Kommandozeilenanwendung behandelt werden.

5.6. Umwandlung

Wir betrachten zunächst die Behandlung von LTL-Formeln und ω-regulären Ausdrücken. Die entsprechenden abstrakten Klassen `Expression` der Pakete `ltl` und `omegaregex` schreiben jeweils eine Methode `toRltl` mit dem Rückgabetyp `Expression` aus dem Paket `rltl` vor. In den Implementierungen dieser Methoden kann ein Operator jeweils die entsprechenden Umwandlungsmethoden auf seine Operanden aufrufen und anschließend die Ergebnisse zu einem RLTL-Ausdruck

kombinieren. Auf diese Weise können die iterativ gegebenen mathematischen Umwandlungsfunktionen von LTL-Ausdrücken und ω-regulären Ausdrücken zu RLTL-Ausdrücken direkt in Scala umgesetzt werden.

Ganz ähnlich wird die Datenstruktur der Formeln auch bei der Umwandlung von regulären Ausdrücken und RLTL-Ausdrücken in Automaten ausgenutzt. Dafür schreibt die abstrakte Klasse `Expression` aus dem Paket `rltl` eine Methode `toApws` und die abstrakte Klasse `Expression` aus dem Paket `regex` eine Methode `toNfa` vor. Beide Verfahren sind konstruktive Bottom-Up-Umwandlungen, sodass ein Operator jeweils die Umwandlung seiner Operanden ausführen und die so entstandenen Automaten anschließend zu einem neuen Automaten zusammensetzen kann. Diese rekursiven Aufrufe befinden sich jeweils in den Implementierungen dieser abstrakten Methoden. Das Zusammensetzen der Automaten zu größeren Automaten übernehmen dann allerdings Operatoren in den Klassen `Apw` und `Nfa`. Durch diese Aufteilung können die Implementierungsdetails von Formeln und Automaten möglichst gut voneinander getrennt werden, sodass diese auch unabhängig voneinander geändert werden können. Da die Umwandlungsfunktion für RLTL auf Paaren von APWs arbeitet, existiert die Klasse `ApwSpecular`, die zwei Instanzen der Klasse `Apw` speichert. In dieser Klasse sind Operatoren definiert, mit denen direkt Paare von APWs zu einem neuen Paar von APWs kombiniert werden können. Diese Operatoren nutzen dabei intern wieder Operatoren auf der Klasse `Apw`.

Die Umwandlung der RLTL-Power-Operatoren wird hier exemplarisch näher betrachtet. Die Klasse `Power` aus dem Paket `rltl` besitzt folgende Methode:

```
def toApws(implicit alphabet: Set[Sign]) =
  obligation.toApws power(delay.toNfa, attempt.toApws)
```

Genauso werden von entsprechenden Methoden in der Klassen `WeakPower`, der Klasse `UniPower` und der Klasse `UniWeakPower` die Operatoren `weakPower`, `uniPower` und `uniWeakPower` aufgerufen. In allen diesen Operatoren wird die Hilfsfunktion `powerBuilder` verwendet, um zwei APWs zu generieren. Die beiden generierten APWs werden dann in einer Instanz von `ApwSpecular` zurückgegeben. Diese Hilfsfunktion erhält zwei APWs, einen NFA, einen positiven booleschen Operator und eine Farbe als Argumente und liefert einen APW zurück. Damit wird gerade der parametrisierte 2APW aller Power-Operatoren, der in Unterabschnitt 4.2.5 auf Seite 58 beschrieben wird, realisiert. Eine ganz analoge Hilfsfunktion existiert mit der Methode `sequentialBuilder`, die als Parameter einen APW, einen NFA und einen positiven booleschen Operator erhält. Mit dieser Hilfsmethode wird in der Klassen `Sequential` und der Klasse `UniSequential` aus dem Paket `rltl` die Methode `toApws` implementiert.

Entsprechend sind in der Klasse `Nfa` Operatoren zur Verkettung, zur Vereinigung und zur Realisierung des Kleene-Operators vorhanden. Zusätzlich existieren im Kom-

pagnonobjekt `Nfa` Methoden zur Erzeugung von NFAs. Diese akzeptieren entweder genau ein Eingabezeichen bzw. true oder false oder erzeugen Automaten von Basisausdrücken in die Vergangenheit. Die Klasse `Past`, die den allgemeinen Vergangenheits-Operator auf regulären Ausdrücken realisiert, implementiert selbst keine Umwandlung zu NFAs. Stattdessen wird von der Methode `toNfa` in dieser Klasse die Methode `removePast` aufgerufen und auf dem auf diese Weise erzeugten regulären Ausdruck wird dann erneut `toNfa` aufgerufen. Durch die Methode `removePast` werden rekursiv alle allgemeinen Vergangenheits-Operatoren durch Vergangenheits-Operatoren auf Basisausdrücken ersetzt. So wird die Umwandlung von regulären Ausdrücken mit allgemeinen Vergangenheits-Operatoren nach außen transparent auf die Umwandlung von Vergangenheits-Operatoren auf Basisausdrücken zurückgeführt.

Bei der mathematischen Beschreibung der Umwandlungen im letzten Kapitel wurde immer davon ausgegangen, dass implizit ein Alphabet gegeben ist. Da die Klassen, aus denen die Formeln bestehen, das Alphabet nicht selbst speichern, muss das Alphabet aus der Instanz der Klasse `Formula` durch alle Rekursionsschritte der Umwandlung durchgereicht werden, sodass immer das Zielalphabet zur Verfügung steht. Um dieses Problem zu lösen, besitzen die Methoden `toApws` bzw. `toNfa` einen impliziten Parameter `alphabet`. Als impliziter Parameter wird `alphabet` automatisch an alle rekursiven Aufrufe der Methoden weitergereicht. Wie in der textuellen Repräsentation dürfen true und false auch hier nicht explizit als Elemente dieses Alphabets mit angegeben werden.

Wie wir im letzten Kapitel gesehen haben, muss ein NFA, bevor er in einen APW eingebaut werden kann, entsprechend vorbereitet werden. Dafür wird folgende Methode aus der Klasse `Nfa` von den Operatoren im der Klasse `Apw`, die einen NFA verwenden, aufgerufen.

```
def toPreparedNfa =
  toReducedNfa.
    toCompletedNfa
```

Die Methode `toReducedNfa` entfernt dabei alle ε-Transitionen. Im nächsten Schritt wird von der Methode `toCompletedNfa` der NFA wie in Unterabschnitt 3.1.3 auf Seite 27 beschrieben zu einem totalen Automaten erweitert. Mit der Hilfsmethode `findJunkState` wird in dieser Methode überprüft, ob bereits eine Senke existiert. Wurde ein solcher Zustand gefunden, so wird dieser als Ziel für die zu ergänzenden Transitionen verwendet, sonst wird eine neue Senke hinzugefügt.

Die Entfernung aller ε-Transitionen besteht wiederum aus zwei Teilschritten.

```
def toReducedNfa =
  toNfaWithoutEpsilon.
    toNfaWithoutUnreachableStates
```

In der Methode `toNfaWithoutEpsilon` werden mit dem in Unterabschnitt 4.1.6 auf Seite 45 beschriebenen Verfahren alle ε-Transitionen entfernt. Dabei wird die partielle Funktion t, die für jeden Zustand mit ausgehenden ε-Transitionen die Ziele dieser Transitionen angibt, durch den Datentyp

```
Map[State,List[Pair[State,Int]]]
```

dargestellt. Mit der Hilfsmethode `closure` wird dann über dieser Tabelle die Hülle gebildet. Die anschließende Behandlung von Zielzuständen mit einer geringeren Akzeptanzbedingung als von der Tabelle gefordert wird über eine doppelte Reduktion realisiert. Mit der Methode `foldLeft` aus der Scala-API wird über die Tabelle iteriert und dabei als Startwert der aktuelle Automat und die Tabelle übergeben. So können in jedem Schritt der Iteration die Tabelle und der Automat angepasst werden. Auf gleiche Weise wird nun für jeden Eintrag der Tabelle über alle Zustände dieses Eintrags iteriert, um für jeden Zustand die benötigten Anpassungen vornehmen zu können. Die so erzeugte Tabelle wird nun auf die Menge der initialen Zustände und die Transitionstabelle angewendet. Dabei werden alle ε-Transitionen entfernt, für jede Transition zu einem Zustand mit einem Eintrag in der Tabelle werden Transitionen zu allen Zuständen in diesem Eintrag hinzugefügt und die Hilfsfunktion `keepIfNeeded` fügt den originalen Zielzustand der Transition wieder hinzu, wenn dieser bleibende ausgehende Transitionen besitzt. Schließlich werden in der Methode `toNfaWithoutUnreachableStates` alle nicht erreichbaren Zustände aus dem Automaten entfernt. Dabei fügt eine Breitensuche alle von den initialen Zuständen aus erreichbaren Zustände der veränderlichen Variablen `reachables` hinzu. Diese ist in diesem Fall keine Liste, sondern eine Menge, sodass möglichst effizient überprüft werden kann, ob ein Zustand bereits besucht wurde. Zurückgegeben wird ein NFA, in dem sich nur die Zustände befinden, die in diese Menge aufgenommen wurden.

Nach der Umwandlung in einen APW stehen verschiedene Optimierungen zur Verfügung. Die in Unterabschnitt 4.4.4 auf Seite 71 beschriebenen zwei Stufen der Optimierung werden dabei durch die folgenden beiden Methoden der öffentlichen Schnittstelle der Klasse `Apw` realisiert.

```
def toSlimApw =
  toSmallerApw.
    toApwWithoutUnreachableStates.
    toApwWithoutFalse

def toReducedApw =
  toSmallerApw.
    toApwWithoutEpsilon.
    toApwWithoutUnreachableStates.
```

```
toApwWithoutFalse.
toApwWithoutTrue.
toNamedApw(forceNewNames = true)
```

Die erste Methode optimiert den Automaten soweit, dass dieser möglichst übersichtlich wird, und die zweite optimiert den Automaten weiter, sodass dieser möglichst leicht von anderen Programmen verwendet werden kann. Das in Unterabschnitt 4.4.3 auf Seite 69 beschriebene Verfahren zur Entfernung aller nicht benötigter Transitionen wird dabei in der Methode `toSmallerApw` realisiert. Dieses wird praktisch durch eine Breitensuche realisiert, die für jede Transition alle Zustände in der positiven booleschen Formel besucht. Für jede Formel wird dabei versucht, eine einfachere Formel ohne true und false zurückzugeben und für jeden Zustand wird nach Möglichkeit true oder false zurückgegeben, wenn ein Übergang in diesen Zustand unabhängig vom Eingabewort immer zum Akzeptieren bzw. Verwerfen der Eingabe führt. Damit die Breitensuche abbricht, werden die Ergebnisse der Aufrufe in der veränderlichen Variablen `results` gespeichert. Existiert für einen Zustand bereits ein Eintrag in dieser Tabelle, so wird dieser als Ergebnis verwendet und die Rekursion bricht ab. Die Methoden `toApwWithoutEpsilon` und `toApwWithoutUnreachableStates` funktionieren prinzipiell genauso wie beim NFA mit dem Unterschied, dass die Ziele von Transitionen nun nicht als Liste von Zuständen, sondern als positive boolesche Formel von Zuständen gespeichert werden. Entsprechend wird die partielle Funktion t, die für jeden Zustand mit ausgehenden ε-Transitionen die Ziele dieser Transitionen angibt, nun durch den Datentyp `Map[State,PosBool[Pair[State,Int]]]` dargestellt. An dieser Stelle wird die Flexibilität des generischen Datentyps `PosBool` in vollem Umfang benötigt. Die Methode `toApwWithoutFalse` gibt einfach einen APW zurück, in dem nur die Transitionen übernommen wurden, die nicht zu false führen. Die Methode `toApwWithoutTrue` prüft hingegen zunächst, ob bereits ein Zustand vorhanden ist, der true repräsentieren kann. Wenn nicht, wird ein solcher Zustand hinzugefügt. Anschließend werden alle Transitionen zu true durch Transitionen zu diesem Zustand ersetzt. Am Ende der Methode `toReducedApw` wird schließlich die Methode `toNamedApw` aufgerufen. In allen anderen Operationen ist ein solcher Aufruf nicht nötig, da dieser implizit vor der Ausgabe eines Automaten erfolgt. Bei dieser impliziten Benennung der Zustände behalten allerdings alle Zustände, die bereits einen Namen besitzen, diesen. Auf diese Weise kann genauer nachvollzogen werden, was bei den Optimierungsschritten passiert. Durch den Parameter `forceNewNames` wird hier aber allen Zuständen ein neuer Name zugeteilt. Dadurch entsteht ein Automat, indem die Zustände in der Liste aller Zustände aufsteigend nummeriert sind. Dies vereinfacht wiederum die Verarbeitung der Ausgabe durch andere Programme, wie die im Rahmen von [Sch11] realisierte Software.

5.7. Kommandozeilenschnittstelle

Um die bisher beschriebenen Klassen nicht nur als API, sondern auch als eigenständiges Programm nutzen zu können, wurde eine Kommandozeilenanwendung entwickelt. Diese Anwendung soll möglichst einfach zu bedienen sein, aber trotzdem die Vielzahl an Anwendungsmöglichkeiten neben der primären Umwandlung von RLTL-Formeln zu APWs zugänglich machen. Daher wird dem Programm als Kommandozeilenargumente eine Reihe von Umwandlungen mitgegeben, die auf ein Objekt der Reihe nach angewendet werden. Ein solches Objekt ist entweder eine Formel, ein Automat oder ein gespiegeltes Automatenpaar. Vor der ersten Umwandlung muss ein Objekt in textueller Repräsentation eingelesen werden. Das Ergebnis der letzten Umwandlung wird am Ende in textueller Repräsentation ausgegeben. Ist der erste Parameter keine Umwandlung, so wird dieser als Dateiname interpretiert, wenn er mit einem At-Zeichen (`@`) beginnt, oder sonst direkt als Eingabe verwendet. Ist der erste Parameter direkt eine Umwandlung, so wird die Eingabe von der Standardeingabe eingelesen. Die Ausgabe geschieht immer auf der Standardausgabe.

Die Kommandozeilenanwendung befindet sich im Paket `cli` in dem Singleton-Objekt `Main` und dem Singleton-Objekt `Conversion`. Das Objekt `Main` besitzt eine Methode `main` und kann ausgeführt werden. Das Objekt `Conversion` besitzt eine Methode `convert`, der der Name einer Umwandlung als String zusammen mit einem Objekt übergeben wird. Existiert die Umwandlung, wird sie angewendet und das Ergebnis zurückgegeben. Alle Umwandlungen werden als partielle Funktionen in einer Tabelle mit ihrem Namen als Schlüssel gespeichert. Auf diese Weise muss nicht jede Umwandlung jedes beliebige Objekt behandeln können, und bereits im Objekt `Main` kann überprüft werden, ob die aktuelle Umwandlung auf dem aktuellen Objekt definiert ist. Wenn nicht, wird eine entsprechende Fehlermeldung ausgegeben. Schließlich existiert noch eine Tabelle mit impliziten Umwandlungen. Ist eine Umwandlung nicht auf dem aktuellen Objekt definiert, aber ist für diese Umwandlung eine implizite Umwandlung definiert, so wird versucht, diese auf das Objekt anzuwenden und die eigentliche Umwandlung auf das Ergebnis der impliziten Umwandlung anzuwenden. Auf diese Weise können auf textuelle Repräsentationen von Automaten und Formeln direkt inhaltliche Umwandlungen angewendet werden und das Parsen wird implizit ausgeführt. Die folgenden Umwandlungen stehen zur Verfügung:

`--automata` Parst eine textuelle Repräsentation eines Automaten. Diese Umwandlung erfolgt implizit bei den mit (A) markierten Umwandlungen.

`--formula` Parst eine textuelle Repräsentation einer Formel. Diese Umwandlung erfolgt implizit bei den mit (F) markierten Umwandlungen.

`--future` **(F)** Ruft `removePast` auf einen regulären Ausdruck auf. So werden alle allgemeinen Vergangenheits-Operatoren durch Vergangenheits-Operatoren auf Basisausdrücken ersetzt.

`--positive` **(F)** Ruft `removeNegation` auf einen RLTL-Ausdruck auf. Dadurch werden alle Negationsoperatoren entfernt.

`--dot` **(A)** Ruft `toDot` auf einem Automaten auf. Dadurch wird ein Automat in das Dot-Format exportiert.

`--slim` **(A)** Ruft `toSlimApw` auf einem APW auf.

`--reduce` **(A)** Ruft `toReducedApw` auf einem APW und `toReducedNfa` auf einem NFA auf.

`--complete` **(A)** Ruft `toCompletedNfa` auf einem NFA auf. Dadurch wird dieser zu einem totalen Automaten ergänzt.

`--nfa` **(F)** Ruft `toNfa` auf einem regulären Ausdruck auf. Der reguläre Ausdruck wird dabei zu einem NFA umgewandelt.

`--apw` **(F)** Ruft `toApws` auf einem RLTL-Ausdruck auf und gibt nur den ersten Automaten aus dem Paar zurück. So wird ein RLTL-Ausdruck zu einem APW umgewandelt.

`--!apw` **(F)** Ruft `toApws` auf einem RLTL-Ausdruck auf und gibt nur den zweiten Automaten aus dem Paar zurück. So wird ein RLTL-Ausdruck negiert und dann zu einem APW umgewandelt.

`--apws` **(F)** Ruft `toApws` auf einem RLTL-Ausdruck auf und gibt das gespiegelte Automatenpaar zurück. So wird ein RLTL-Ausdruck zu einem APW und einem APW des negierten Ausdrucks umgewandelt.

`--rltl` **(F)** Ruft `toRltl` auf einem LTL-Ausdruck oder auf einem ω-regulären Ausdruck auf. So werden diese zu RLTL-Ausdrücken umgewandelt.

`--profile` Startet die Zeitmessung oder setzt diese zurück.

Statt einem APW kann auch jeweils ein gespiegeltes Paar von APWs behandelt werden. Die Umwandlungen werden dann jeweils auf beide Automaten angewendet und bei der Ausgabe werden beide Automaten getrennt durch einen Leerzeile ausgegeben. Solche Paare von Automaten können nicht wieder importiert werden.

Von der Umwandlung `--profile` wird das aktuelle Objekt nicht verändert. Stattdessen wird eine Zeitmessung gestartet. Wurde im Laufe des Programms eine Zeitmessung gestartet, so wird am Ende der Umwandlunskette nicht das eigentliche Ergebnis der Berechnung, sondern die verstrichene Zeit ausgegeben.

Wird bei einer Umwandlung eine Ausnahme geworfen oder wird für ein Objekt eine Umwandlung verlangt, die dieses Objekt nicht unterstützt, so wird eine Fehlermeldung zusammen mit einer kurzen Anleitung ausgegeben. Intern wird dies über das Werfen einer Ausnahme `CliException` realisiert.

Zur Ausführung des JAR-Archivs `rltl.jar` wird die Scala-Laufzeitumgebung auf dem JAR-Archiv `scala-library.jar` benötigt. Befinden sich beide Dateien im aktuellen Verzeichnis, kann das Programm mit folgendem Befehl aufgerufen werden:

```
java -cp rltl.jar;scala-library.jar cli.Main ARG1 ARG2 ...
```

Als `ARG1` muss dabei eine Eingabe oder eine Umwandlung und als folgende Argumente jeweils eine Umwandlung angegeben werden. Die Batch-Datei `rltl.bat` übernimmt unter Windows das Laden der JAR-Dateien, sodass dieser direkt die Argumente übergeben werden können. Angenommen die Datei `test.rltl` hat folgendem Inhalt:

```
RLTL=a;!% / TRUE TRUE > %, ALPHABET=[a,b]
```

So kann eine Umwandlung der RLTL-Formel

$$\varphi := a; \neg\emptyset / \operatorname{true}\operatorname{true}\rangle\emptyset$$

mit dem Alphabet $\Sigma = \{a, b\}$ in ein Paar von optimierten APWs mit folgendem Kommando erfolgen:

```
rltl @test.rltl --apws --reduce
```

Sollen stattdessen nur leicht optimierte APWs im Dot-Format generiert werden und diese in eine Grafik umgewandelt werden, so können folgende Kommandos verwendet werden.

```
rltl @test.rltl --apw --slim --dot > pos.dot
rltl @test.rltl --!apw --slim --dot > neg.dot
dot -O -Tpng pos.dot
dot -O -Tpng neg.dot
```

Wir erhalten die Grafiken aus Abbildung 5.7 auf der nächsten Seite und Abbildung 5.8 auf der nächsten Seite.

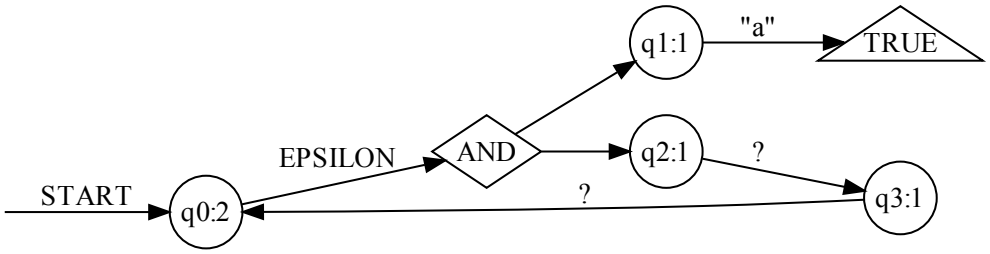

Abbildung 5.7.: Mit dot gezeichneter Graph des APWs der Formel φ über dem Alphabet $\Sigma = \{a, b\}$.

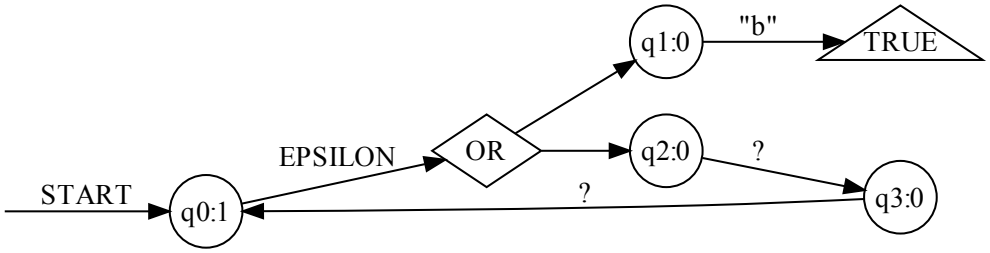

Abbildung 5.8.: Mit dot gezeichneter Graph des APWs der negierten Formel φ über dem Alphabet $\Sigma = \{a, b\}$.

5.8. Tests mit praktischen Beispielen

Matthew Dwyer, George Avrunin und James Corbett haben in [DAC98] LTL-Pattern, die in der Praxis häufig verwendet werden, untersucht. Im Rahmen dieser Arbeit ist eine Liste häufig verwendeter LTL-Formeln entstanden. Aus 97 dieser Formeln wurden im Rahmen dieser Arbeit RLTL-Formeln generiert. Da in der Praxis noch keine RLTL-Formeln existieren, stellen diese 97 RLTL-Formeln die einzigen Testdaten für die Umwandlung von RLTL-Formeln dar. Da die Mächtigkeit von RLTL größer ist als die Mächtigkeit von LTL, decken diese Formeln nicht alle Möglichkeiten von RLTL ab, insbesondere kommt in LTL-Formeln immer nur eine implizite Verzögerung von einem Zeichen vor. Ebenso handelt es sich bei den verwendeten LTL-Formeln nur um Formeln ohne Vergangenheit. Trotz dieser Einschränkungen sind diese Testdaten insoweit relevant, als sie nachgewiesenen Praxisbezug besitzen.

Die Datei `rv_properties.txt` enthält 97 Pattern durch Leerzeilen getrennt. Jedes Pattern besteht aus einer Beschreibung, einer LTL-Formel und einer von LTL-Parsern leicht zu verarbeitenden Form der Formel. Diese Form beginnt jeweils mit einem Bindestrich »-« und kann direkt von der in dieser Arbeit realisierten Kommandozeilenanwendung geparst werden. Dazu erzeugt das Windows-Batch-Script `create_ltl.bat` unter Verwendung der Tools grep und sed für jede Formel eine eigene Eingabedatei. Weitere Scripts verwenden das oben beschriebene Script `rltl.bat` und erzeugen aus den Formeln APWs in textueller Repräsentation, RLTL-Formeln und Ausgaben der Automaten als Grafik unter Verwendung von dot. Die APWs in textueller Repräsentation können von dem im Rahmen von [Sch11] entwickelten Programm eingelesen und zu NBWs umgewandelt werden und die RLTL-Formeln bilden eine Sammlung gängiger RLTL-Formeln, die für weitere Testzwecke in anderen Projekten zur Verfügung stehen. Die Kontrolle der Korrektheit der Umwandlungen konnte nur manuell anhand dieser Grafiken erfolgen, da für eine solche Umwandlung keine Vergleichsdaten zur Verfügung stehen. Dabei hat sich ergeben, dass viele der LTL-Formeln zu sehr trivialen Automaten zusammenfallen, wenn das Alphabet automatisch aus den in der Formel vorkommenden Zeichen erraten wird. Erst mit der expliziten Angabe eines Alphabets, das weitere Zeichen enthält, entstehen sinnvolle Automaten. Eine solche Angabe eines Alphabets fehlt aber in der verwendeten Sammlung von LTL-Formeln und konnte entsprechend nur recht willkürlich zu Testzwecken ergänzt werden.

Die Laufzeit der Umwandlungen lag mit Ausnahme einer sehr langen LTL-Formel immer deutlch unter einer Sekunde. Die Laufzeit kann also im Vergleich zu der Laufzeit der anschließenden Umwandlung in einen NBW vernachlässigt werden, da diese häufig im Bereich von Minuten lag. Eine dedizierte Analyse der Laufzeiten ist allerdings nicht erfolgt, da hierfür zunächst ein genaues Ziel der Untersuchung

definiert werden müsste. So variieren zum Beispiel die Laufzeiten der verschiedenen Umwandlungs- und Optimierungsoperationen von Formel zu Formel stark und auch das Starten der Java Virtual Machine und Laden der Scala-Laufzeitumgebung benötigt bei kurzen eigentlichen Laufzeiten signifikante Zeit. Für die theoretische Betrachtung des Umwandlungsalgorithmus ist natürlich gerade die Betrachtung einer Formel, bei der ein Bestandteil immer häufiger wiederholt wird, von Interesse, um die allgemeine Zeitkomplexität überprüfen zu können. Für die praktische Anwendung haben solche Untersuchungen aber wenig Relevanz. Hier wäre eher zu untersuchen, wie schnell die Bibliothek RLTL-Formeln umwandeln kann, wenn sie in einen Model Checker eingebunden wird. Eine solche Integration ist aber nicht mehr Thema dieser Arbeit.

6. Zusammenfassung und Ausblick

Die im Rahmen dieser Arbeit implementierte Softwarebibliothek ist in der Lage, eine RLTL-Formel in einen 2APW umzuwandeln. Das Ergebnis der Umwandlung kann im eigenen Dateiformat exportiert werden und der Graph des Automaten kann mit Hilfe von Graphviz als Grafik generiert werden. Darüber hinaus ist die Software in der Lage, alle Zwischenschritte dieser Umwandlung auch einzeln auszuführen. So können zum Beispiel reguläre Ausdrücke in NFAs umgewandelt werden, ε-Transitionen aus beliebigen APWs und NFAs entfernt werden oder verschiedene andere Operationen auf APWs und NFAs angewendet werden. Zusätzlich können LTL-Formeln und ω-reguläre Ausdrücke in RLTL-Formeln umgewandelt werden. Zur einfachen Verwendung dieser Softwarebibliothek wurde eine Kommandozeilenanwendung implementiert, mit der viele der implementierten Operationen flexibel angewendet werden können.

Damit RLTL-Formeln in vorhandenen Umgebungen zur Laufzeitverifikation oder Model-Checking-Frameworks verwendet werden können, wird eine Software benötigt, die RLTL-Formeln in NBWs umwandelt. Ursprünglich war es das Ziel dieser Arbeit und [Sch11], eine solche Software zu implementieren. Es hat sich aber herausgestellt, dass die Umwandlung eines 2APWs in einen APW oder die Umwandlung eines 2APWs direkt in einen NBW ein so komplexes Thema ist, dass es nicht in diesen Arbeiten behandelt werden kann. Entsprechend wird für diese Umwandlung noch eine Lösung benötigt, um das skizzierte Ziel zu erreichen. Für RLTL-Formeln ohne Vergangenheit liefert allerdings die im Rahmen dieser Arbeit implementierte Bibliothek direkt einen APW, der auch direkt von der im Rahmen von [Sch11] implementierten Software verwendet werden kann. Entsprechend ist für diese Teilmenge von RLTL-Formeln das Ziel bereits erreicht.

Ebenso wie die Problematik der Zwei-Wege-Automaten wurde zu Beginn dieser Arbeit auch die Behandlung der ε-Transitionen, die beim Erzeugen der NFAs und der APWs aus regulären Ausdrücken bzw. RLTL-Formeln entstehen, nicht ausreichend bedacht. Dieses Problem konnte gelöst werden, indem eine allgemeine Umwandlung von Automaten mit ε-Transitionen zu Automaten ohne ε-Transitionen unter Ausnutzung des Nichtdeterminismus für APWs und NFAs realisiert wurde. Diese Umwandlungen können auch unabhängig von der Umwandlung von RLTL-Formeln in anderen Projekten verwendet werden. Im Nachhinein hat sich allerdings herausgestellt, dass dieser allgemeine Ansatz nicht zwingend notwendig ist. Stattdessen könnte die

Umwandlung von RLTL-Formeln und regulären Ausdrücken auch so umgestaltet werden, dass direkt der Nichtdeterminismus ausgenutzt wird anstatt ε-Transitionen zu erzeugen. Dadurch könnte vermutlich ein gewisser Geschwindigkeits-Gewinn erzielt werden, da ein Optimierungsschritt wegfällt. Es wäre dann allerdings auch nicht mehr möglich, den deutlich besser lesbaren Automaten mit ε-Transitionen auszugeben.

Die realisierte Softwarebibliothek wurde im Rahmen dieser Arbeit nicht in bestehende Laufzeitverifikations-Frameworks oder Model-Checking-Tools integriert. Eine solche Integration stellt neben der Behandlung von 2APWs den nächsten Schritt auf dem Weg zur praktischen Anwendung von RLTL-Formeln dar. Um nachzuweisen, dass die Umwandlung von RLTL-Formeln zu NBWs funktioniert, wurden ca. 100 praktisch relevante LTL-Formeln aus [DAC98] in RLTL-Formeln umgewandelt. Somit steht nun auch für andere Projekte erstmals eine Sammlung von praktisch relevanten RLTL-Formeln zu Verfügung. Diese Formeln wurden in APWs umgewandelt und diese wiederum in NBWs. Eine manuelle Überprüfung dieser APWs und NBWs konnte die Korrektheit der Implementierungen bestätigen. Ein Vergleich mit anderen Umwandlungen von LTL-Formeln in NBWs steht noch aus.

Neben der Integration in bestehende Software kann auch die Idee weiter verfolgt werden, die im Rahmen dieser Arbeit und im Rahmen von [Sch11] realisierte Software als Webservice anzubieten. Für die Realisierung dieses Webservices bietet sich das Scala-Webframework Lift[1] an, da so bestehenden Scala- und Java-Bibliotheken direkt verwendet werden können. Ein solcher Webservice könnte sehr einfach in anderen Projekten eingebunden werden ohne lokal Software installieren zu müssen. Daneben könnte eine Weboberfläche angeboten werden, auf der RLTL-Formeln in Automaten umgewandelt werden können. Eine Weboberfläche könnte darüber hinaus die Verwendung von dot vereinfachen und das Ergebnis direkt als Bild zur Verfügung stellen. Neben der endgültigen Ausgabe könnte eine Weboberfläche auch die verschiedenen Zwischenschritte darstellen oder die verwendeten Operationen erläutern und auf diesem Wege das Verständnis für die beteiligten Formelarten und Automatentypen erhöhen.

Im Rahmen dieser Arbeit wurde also ein umfangreicher Werkzeugkasten für die Arbeit mit RLTL-Formeln realisiert, der alle in [LS07], [SL10] und [SSF11] angesprochenen Algorithmen für die Verwendung von RLTL implementiert. Damit können RLTL-Formeln in alternierende Paritätsautomaten umgewandelt werden, sodass RLTL in der Praxis zur Laufzeitverifikation und der Modellprüfung eingesetzt werden kann.

[1] siehe `http://liftweb.net/`

A. Verzeichnisse

A.1. Literaturverzeichnis

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett.
Property specification patterns for finite-state verification.
In Mark A. Ardis and Joanne M. Atlee, editors, *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15. Association for Computing Machinery (ACM), 1998.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, second edition, 2001.

[LS07] Martin Leucker and César Sánchez.
Regular Linear Temporal Logic.
In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Proceedings of the 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, volume 4711 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2007.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners.
Programming in Scala: A Comprehensive Step-by-step Guide.
Artima Incorporation, first edition, 2008.

[Rac09] Meiko Rachimow.
Evaluierung des Einsatzes von Scala bei der Entwicklung für die Android-Plattform.
Diplomarbeit an der technischen Fachhochschule Berlin, 2009.

[Sch11] Torben Scheffel.
Transformation von Paritätsautomaten in Büchi-Automaten.
Bachelorarbeit an der Universität zu Lübeck, 2011.

[She59] J. C. Shepherdson.
The reduction of two-way automata to one-way automata.
IBM Journal of Research and Development, 3:198–200, 1959.

[SL10] César Sánchez and Martin Leucker.
Regular Linear Temporal Logic with Past.
In Gilles Barthe and Manuel Hermenegildo, editors, *Proceedings of the 11th International Conference on Verification, Model Checking and*

Abstract Interpretation (VMCAI'10), volume 5944 of *Lecture Notes in Computer Science*, pages 295–311. Springer, 2010.

[Spi11] Daniel Spiewak.
The Magic Behind Parser Combinators.
`http://www.codecommit.com/blog/scala/the-magic-behind-parser-combinators`, 2011.
[Online; Zugriff am 30.11.2011].

[SSF11] César Sánchez and Julián Samborski-Forlese.
Efficient Regular Linear Temporal Logic using Dualization and Stratification.
bisher unveröffentlicht, 2011.

[Var89] Moshe Y. Vardi.
A note on the reduction of two-way automata to one-way automata.
Information Processing Letters, 30:261–264, 1989.

A.2. Abkürzungsverzeichnis

RLTL Reguläre Linearzeit-Temporallogik,
engl. *R*egular *L*inear *T*emporal *L*ogic

LTL Linearzeit-Temporallogik,
engl. *L*inear *T*emporal *L*ogic

EBNF Erweiterte Backus-Naur-Form,
engl. *E*xtended *B*ackus–*N*aur *F*orm

DFA deterministischer endlicher Automat auf endlichen Worten,
engl. *D*eterministic *F*inate State *A*utomaton

NFA nichtdeterministischer endlicher Automat auf endlichen Worten,
engl. *N*ondeterministic *F*inite State *A*utomaton

UFA universeller endlicher Automat auf endlichen Worten,
engl. *U*niversal *F*inite State *A*utomaton

2NFA nichtdeterministischer endlicher Zwei-Wege-Automat auf endlichen Worten,
engl. *2*-way *N*ondeterministic *F*inite State *A*utomaton

2UFA universeller endlicher Zwei-Wege-Automat auf endlichen Worten,
engl. *2*-way *U*niversal *F*inite State *A*utomaton

NBW nichtdeterministischer endlicher Büchi-Automat auf unendlichen Worten,
engl. *N*ondeterminitic Finite State *B*üchi Automaton on *W*ords

APW endlicher alternierender Paritätsautomat auf unendlichen Worten,
engl. *A*lternating *P*arity Finite State Automaton on *W*ords

2APW endlicher alternierender Zwei-Wege-Paritätsautomat auf unendlichen Worten, engl. *2*-way *A*lternating *P*arity Finite State Automaton on *W*ords

DAG gerichteter azyklischer Graph,
engl. *D*irected *A*cyclic *G*raph

AFF Automatendateiformat,
engl. *A*utomata *F*ile *F*ormat

A.3. Korrekturverzeichnis

Die folgenden Korrekturen bzw. Hinweise wurden *nach* der Abgabe und Bewertung der Arbeit ergänzt.

- In Abbildung 4.24 auf Seite 64 und Abbildung 4.25 auf Seite 65 wurden die doppelten Umrandungen einiger Zustände entfernt. In den Abbildungen werden 2APWs dargestellt, die diese Syntax nicht vorsehen, da sie keine akzeptierenden Zustände haben.
- Die Formel $p/\,\mathrm{true}\,\mathrm{true}\rangle\,\mathrm{false}$ aus dem Beispiel in Unterabschnitt 2.3.2 auf Seite 16 wurde zu $p;\neg\emptyset/\,\mathrm{true}\,\mathrm{true}\rangle\emptyset$ geändert. p und false sind (alleine) keine gültigen RLTL-Ausdrücke.
- Die LTL-Grammatik in Unterabschnitt 2.4.1 auf Seite 20 akzeptierte unter anderem Ausdrücke der Form $\bigcirc\bigcirc a$ oder $p\,\mathcal{U}\,q\,\mathcal{U}\,r$ nicht. Daher wurden die Zeilen `B = U "𝒰" U | U` der binären Operatoren zu `B = U "𝒰" B | U` korrigiert und die Zeile `U = "◯" P | P` der unären Operatoren zu dem neuen Ausdruck `U = "◯" U | P` korrigiert. Der Next- und Until-Operator steht dabei nur die stellvertretend für alle Operatoren dieser Zeilen. Damit entspricht die Grammatik nun auch wieder der Implementierung, in der dieser Fehler bereits korrigiert wurde. Entsprechend wurde auch das Beispiel gegen Ende von Abschnitt 5.4 auf Seite 85 über das effiziente Parsen von Ausdrücken der Art `(((((((((c)))))))))` korrigiert.
- In Unterabschnitt 5.5.1 auf Seite 95 wurde die Bezeichnung Automatendateiformat (AFF) für die im Rahmen dieser Arbeit entworfene textuelle Repräsentation von Automaten eingeführt. AFF wurde ebenfalls in das Abkürzungsverzeichnis (Abschnitt A.2 auf der vorherigen Seite) aufgenommen.
- Die Kopfzeile wurde so angepasst, dass nur dann zwei Elemente ausgegeben werden, wenn auch tatsächlich eine aktiver Abschnitt vorhanden ist. Sonst wird nur das aktuelle Kapitel ausgegeben. Die Kopfzeile im Anhang wurde ebenfalls korrigiert.
- Die grafische Darstellung der LTL-Symbole wurden überarbeitet.
- Die Semantik der LTL-Symbole wurde leicht angepasst: Der Strong-Previous-Operator wurde entfernt. Dafür wurde der Weak-Previous-Operator eingeführt und der Previous-Operator wurde entsprechend angepasst. Die Operatoren Since und Back wurden umsortiert, um die Dualität von Until und Since deutlicher darzustellen. Der Operator Trigger wurde als Release in die Vergangenheit eingeführt.

A.4. Abbildungsverzeichnis

A.5. Tabellenverzeichnis