Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 28
D-70569 Stuttgart

Studienarbeit Nr. 2302

Adaptives Model-Checking Reaktiver Systeme

Andreas Voetter

Studiengang: Informatik

Prüfer: Dr. habil. Dirk Nowotka

Betreuer: Dr. habil. Dirk Nowotka

Begonnen am 16.08.2010

Beendet am: 30.09.2010

CR-Klassifikation: F.2.2

Inhaltsverzeichnis

1 Einleitung

In dieser Studienarbeit werden zwei unterschiedliche Bereiche aus einerseits dem Gebiet des maschinellen Lernen und anderseits aus dem Gebiet der Softwarezuverlässigkeit dargestellt. Aus dem Bereich des maschinellen Lernens werden zwei Algorithmen vorgestellt, namentlich der $\mathcal{L}^*$-Algorithmus und der $\mathcal{L}^\omega$-Algorithmus . Für den Zweiten wurde im Rahmen dieser Arbeit eine konkrete Implementierung aus gefertigt. Der $\mathcal{L}^*$-Algorithmus verbindet die beiden fachlichen Gebiete miteinander, da dieser in dem Kapitel 6 mit eingesetzt wird, um das dort beschriebene Adaptive Modell Checking Verfahren zu implementieren.

2 Theoretische Grundlagen

Ein ω-Wort bzw. ω-Sequenz $a \in \Sigma^\omega$ kann als eine Abbildung von $a : \mathbb{N} \to \Sigma$ angesehen werden. Es gibt einige Ähnlichkeiten wie auch Unterschiede zwischen Σ^* und Σ^ω, von denen wir nun einige betrachten wollen. Zum Beispiel ist die sequenzielle Konkatenation $\Sigma^\omega \times \Sigma^\omega$ nicht definiert, sehr wohl aber $\Sigma^* \times \Sigma^\omega$. Jedes $a \in \Sigma^\omega$ hat unendlich viele Zerlegungen der Form $a = u \cdot \beta$ in einen endlichen Präfix a und einen unendlichen Suffix β. Für ein endliches Wort u definieren wir u^ω als die unendliche Aneinanderreihung von $u's$. Wir nennen ein $a \in \Sigma^\omega$ *ultimativ periodisch*, genau dann wenn es eine Zerlegung $a = u\beta^\omega$, wobei u als der Präfix und β als die Periode von a bezeichnet wird. Des weiteren gilt zu beachten, dass für ω-Wörter a, b durchaus $a \in Suffix(b)$ und $b \in Suffix(a)$ gelten kann. Es gibt verschiedene Möglichkeiten, eine endliche und eine unendliche Mengen miteinander in Verbindung zu bringen:

Definition 1. *Für alle $U, V \subseteq \Sigma^*$ und $W \subseteq \Sigma^\omega$*

1. *$V^\omega \subseteq \Sigma^\omega$ bezeichnet die Menge aller ω-Sequenzen $\alpha = v_1 v_2 \ldots$ mit $\forall v_i \in V$.*
2. *$UW \subseteq \Sigma^\omega (U \subseteq \Sigma^*, W \subseteq \Sigma^\omega)$ bezeichnet die Menge aller ω-Sequenzen $\alpha = u\beta$ mit $u \in U$ und $\beta \in W$.*
3. *$\lim U \subseteq \Sigma^\omega$ bezeichnet die Menge aller ω-Sequenzen, welche unendlich viele Präfixe in U haben.*

Definition 2. *Die **ultimativ periodischen Wörter** sind die Wörter der Form $\alpha\beta^\omega \in \Sigma^\omega$. Interessant ist diese Klasse von Wörter, durch die Möglichkeit ihrer endlichen Repräsentierung und der Tatsache, dass für die Durchführung des $\mathcal{L}^\omega$-Algorithmus eben genau diese Wörter ausreichend sind.*

Definition 3. *Ein **minimaler adäquater Lehrer** bezeichnet einen Lehrer, der in der Lage ist zwei verschieden Anforderungen zu erfüllen. Zum einem die Membership-Anfrage, die für einen übergebenes Wort w mit JA bzw. $NEIN$ antwortet, dementsprechend ob $w \in \mathcal{U}$ gilt.*

Zum Zweiten verfügt ein solcher Lehrer über die Conjecture-Anfrage, die für eine übergebene Beschreibung einer Menge $\mathcal{M}$ entscheidet, ob diese äquivalent mit $\mathcal{U}$ ist und in diesem Fall mit JA antwortet und ansonsten ein Gegenbeispiel aus der symmetrischen Differenz von $\mathcal{M}$ und $\mathcal{U}$ liefert.

Definition 4. $\mathcal{B}$ und $\bar{\mathcal{B}}$-Automaten *unterscheiden sich syntaktisch nicht von einem DEA $(\Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F})$, wobei Σ das zugrunde liegende Alphabet, $\mathcal{Q}$ die Zustandsmenge, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ die Überführungsfunktion und q_0 den Startzustand bezeichnet. In dieser Arbeit betrachten wir ausschließlich deterministische Automaten.*

Der Unterschied zwischen DEA's und $\mathcal{B}$ und $\bar{\mathcal{B}}$-Automaten beruht auf den unterschiedlichen Akzeptanzbedingungen. Im Gegensatz zu DEA's die dem Erkennen von endlichen Wörtern dienen, sind die Akzeptanzbedingungen für $\mathcal{B}$ und $\bar{\mathcal{B}}$-Automaten auf unendliche Wörter ausgelegt. Für ein unendliches Wort $\alpha \in \Sigma^\omega$ bezeichnet ***Inf(α)*** die Menge der Zustände eines Automaten, die beim Lesen des Wortes α unendlich oft besucht werden.

Definition 5. *$\mathcal{B}$ und $\bar{\mathcal{B}}$-Akzeptanz: Sei $\mathcal{A} = (\Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F})$ ein Automat. Man spricht von $\mathcal{B}$-Akzeptanz, wenn die von ihm erkannte Menge von Wörtern wie folgt definiert ist:*

$$\mathbb{L}_A = \{\alpha \in \Sigma^\omega : Inf(\alpha) \cap \mathcal{F} \neq \emptyset\} \tag{1}$$

Die $\bar{\mathcal{B}}$-Akzeptanz ist wie folgt definiert:

$$\mathbb{L}_A = \{\alpha \in \Sigma^\omega : Inf(\alpha) \subseteq \mathcal{F}\} \tag{2}$$

Die erste Akzeptanzbedingung erkennt alle Wörter, welche unendlich oft einen Endzustand besuchen. Die Zweite hingegen, erkennt alle die Wörter, die ab einem beliebigen Punkt ab, nur noch Endzustände besuchen.

Interessant sind für den $\mathcal{L}^\omega$-Algorithmus genau die Sprachen, welche sich durch $\mathcal{B}$ und $\bar{\mathcal{B}}$-Automaten erkennen lassen.

Satz 1. *Für alle Sprachen aus $\mathcal{B} \cap \bar{\mathcal{B}}$ existiert immer ein Automat, so dass entweder alle oder keiner der Zustände einer maximalen Zusammenhangskomponenten (MSCC) in $\mathcal{F}$ liegen.*

3 Der $\mathcal{L}^*$-Algorithmus

In diesem Kapitel wird der $\mathcal{L}^*$-Algorithmus vorgestellt, welcher eine anfangs unbekannte reguläre Menge $\mathcal{U}$ über einem vorgegebenen Alphabet Σ in effizienter Weise von einem *minimalen adequaten Lehrer* lernt. Hierfür verwaltet der $\mathcal{L}^*$-Algorithmus eine *Observation Table*. Die Beweisstruktur in dieser Arbeit lehnt sich an die aus dem Originalpaper [D.Angluin] an.

3.1 Observation Table

Definition 6. *Die Observation Table beinhaltet alle Informationen, die der $\mathcal{L}^*$-Algorithmus während seiner Laufzeit über $\mathcal{U}$ sammelt. Eine OT ist ein Dreier-Tupel (S, E, T), wobei S eine unter Präfix abgeschlossene Menge und E eine unter Suffix abgeschlossene Menge bezeichnet. T ist eine Funktion von $((S \cup S \cdot \Sigma) \cdot E)$ nach $\{0, 1\}$, so dass gilt: $T(u) = 1 \Leftrightarrow u \in \mathcal{U}$.*

Zu Beginn des Algorithmus ist die *OT* leer, das heißt $S = E = \{\lambda\}$ und wird mit jedem Durchgang von L^* erweitert. Eine *OT* kann als zweidimensionale Tabelle betrachtet werden, deren Spalten mit den Elementen aus E und die Reihen mit Elementen aus $S \cup S \cdot \Sigma$ beschriftet sind. Für einen Eintrag aus der Reihe $s \in (S \cup S \cdot \Sigma)$ und der Spalte $e \in E$ ergibt sich der Wert durch $T(s \cdot e)$. Für ein $s \in (S \cup S \cdot \Sigma)$ bezeichnet $row(s)$ die endliche Funktion f von E nach $\{0, 1\}$, mit $f(e) = T(s \cdot e)$.

Definition 7. *Closed und consistent Observation Tables. Eine OT wird als closed bezeichnet, wenn für alle $t \in S \cup \Sigma$ ein $s \in S$ existiert, so dass $row(t) = row(s)$ gilt.*

Eine OT wird als consistent bezeichnet, wenn $\forall s_1, s_2 \in S$ mit $row(s_1) = row(s_2) \Rightarrow \forall a \in \Sigma\ row(s_1 \cdot a) = row(s_2 \cdot a)$ gilt.

Wir definieren für jede *closed consistent OT* einen endlichen Automaten $M(S, E, T) = (Q, \Sigma, F, \delta, q_0)$ in folgender Weise:

$$\begin{aligned} Q &= \{row(s), s \in S\} \\ q_0 &= row(\lambda) \\ F &= \{row(s) | s \in S \text{ und } T(s) = 1\} \\ \delta(row(s), a) &= row(s \cdot a) \end{aligned}$$

Zu zeigen bleibt, dass dieser Automat wohldefiniert ist. Da $\lambda \in S \wedge \lambda \in E$ gilt, ist $row(\lambda)$ definiert und damit auch q_0. Somit ist auch Q nicht leer. Da $\lambda \in E$ ist, gilt für s_1, s_2 mit $row(s_1) = row(s_2)$, dass $T(s_1) = T(s_1 \cdot \lambda) = T(s_2 \cdot \lambda) = T(s_2)$, woraus folgt, dass F wohldefiniert ist.

Um zu zeigen, dass δ wohldefiniert ist, gehen wir wieder von s_1, s_2 mit $row(s_1) = row(s_2)$ aus. Da die *OT consitent* ist, gilt für alle $a \in \Sigma$, $row(s_1 \cdot a) = row(s_2 \cdot a)$ und da sie ebenfalls *closed* ist, gibt es auch ein $s \in S$ mit $row(s) = row(s_1 \cdot a) = row(s_2 \cdot a)$.

Theorem 1. *Für eine closed und consistent $OT(S, E, T)$, ist auch der Automat $M(S, E, T)$ mit T konsistent. Jeder andere mit T konsistente, aber zu $M(S, E, T)$ inäquivalente Automat hat mehr Zustände als $M(S, E, T)$.*

Dies werden wir im Folgenden durch eine Reihe von Lemmas beweisen.

Lemma 1. *Sei (S, E, T) eine closed, consistent OT, dann gilt für den Automat $M(S, E, T)$ und für jedes s aus $(S \cup S \cdot A), \delta(q_0, s) = row(s)$*

Dieses Lemma lässt sich leicht per Induktion beweisen. Für den Fall, dass die Länge von $s = 0$ ist, gilt: $s = \lambda$ und laut Definition von $M(S, E, T)$ ist $q_0 = row(\lambda)$.

Sei die Behauptung für alle s der Länge k bewiesen und sei $t = s \cdot a \in (S \cup S \cdot A), a \in \Sigma$ der länge $k + 1$. Da S präfix-abgeschlossen ist, muss s aus S sein. Dann gilt:

$$\begin{aligned}
\delta(q_0, t) &= \delta(\delta(q_0, s), a) \\
&= \delta(row(s), a) && \text{durch die Induktionsannahme} \\
&= row(s \cdot a) && \text{durch die Definition von } \delta \\
&= row(t) && \text{da } t = s \cdot a
\end{aligned}$$

□

Lemma 2. *Sei (S, E, T) eine closed, consistent OT, dann ist der Automat $M(S, E, T)$ konsistent mit der Funktion T. Das heißt, es gilt für alle $s \in (S \cup S \cdot \Sigma)$ und e in E, $\delta(q_0, s \cdot e)$ ist in F genau dann wenn $T(s \cdot e) = 1$.*

Dieses Lemma wird wieder durch Induktion über die Länge von e bewiesen. Wenn e gleich λ ist und s ein Element aus $(S \cup S \cdot \Sigma)$ ist, dann gilt aus dem vorausgegangenem Lemma, dass $(\delta(q_0, s \cdot e) = row(s)$. Wenn s aus S ist, dann gilt durch Definiton von F, dass $row(s)$ in F ist, genau dann wenn $T(s) = 1$. Sollte s aus $(S \cup \Sigma)$ sein, dann gilt da die *OT closed* ist, dass es ein $s_1 \in S$ gibt, mit $row(s) = row(s_1)$ und $row(s_1)$ ist in F genau dann wenn $T(s_1) = 1$ was wiederum gleichbedeutend mit $T(s) = 1$ ist.

Angenommen die Behauptung gilt für alle $d \in E$ der Länge kleiner gleich k und sei $e \in E$ ein Element der Länge $k + 1$. Da E Suffix-abgeschlossen ist, gilt $e = a \cdot e_1$ für $a \in \Sigma, e_1 in E$. Sei s ein beliebiges Element aus $(S \cup S \cdot \Sigma)$. Da die *OT closed* und *consistent* ist, existiert ein String s_1 in S, so dass $row(s) = row(s_1)$ ist ($s = s_1$ ist auch möglich). Es folgt:

$$\begin{aligned}
\delta(q_0, s \cdot e) &= \delta(\delta(q_0, s), a \cdot e_1) \\
&= \delta(row(s), a \cdot e_1), && \text{durch vorhergehendes Lemma} \\
&= \delta(row(s_1), a \cdot e_1), && \text{da } row(s) = row(s_1) \\
&= \delta(\delta(row(s_1), a), e_1) \\
&= \delta(row(s_1 \cdot a), e_1) && \text{durch Definition von } \delta \\
&= \delta(\delta(q_0, s_1 \cdot a), e_1) && \text{durch vorhergehendes Lemma} \\
&= \delta(q_0, s_1 \cdot a \cdot e_1)
\end{aligned}$$

Nun kann man die Induktionsannahme auf e_1 anwenden und erhält $\delta(q_0, s_1 \cdot a \cdot e_1)$ ist in F genau dann wenn $T(s_1 \cdot a \cdot e_1) = 1$. Da $row(s) = row(s_1)$ und $a \cdot e_1 = e$ in E ist, gilt: $T(s_1 \cdot a \cdot e_1 = T(s \cdot a \cdot e_1) = T(s \cdot e)$. Daraus folgt wie behauptet, dass $\delta(q_0, s \cdot e) \in F \Leftrightarrow T(s \cdot e) = 1$. $\square$

Lemma 3. *Sei (S, E, T) eine closed, consistent OT. Angenommen $M(S, E, T)$ hat n Zustände. Wenn ein Automat $M' = (\mathcal{Q}', q_0', F', \delta')$ existiert, der ebenfalls mit T konsistent ist und n oder weniger Zustände besitzt, dann ist M' isomorph zu $M(S, E, T)$*
.

Wir beweisen dieses Lemma durch aufstellen eines Isomorphismus. Definiere für jeden Zustand q' in $\mathcal{Q}'$, $row(q')$ als die Funktion f von E nach $0, 1$ so dass $f(e) = 1$ genau dann wenn $\delta'(q', e)$ in F' ist.

Da M' mit T konsistent ist, gilt $\delta'(\delta'(q_0', s), e) \in F$ genau dann wenn $T(s \cdot e$ gilt. Also ist $row(\delta'(q_0', s))$ gleich $row(s)$ aus $M(S, E, T)$. Da nun aber $row(s)$ genau die Zustände in $M(S, E, T)$ repräsentieren, muss M' mindestens genau so viele Zustände haben, da aber die Annahme ist, dass M' maximal n Zustände besitzt, folgt daraus, dass M' genau n Zustände besitzt.

Wir können also jedem Zustand $row(s)$ aus $M(S, E, T)$ einen Zustand q aus M' zuordnen, in dem wir $\phi(row(s)) = \delta'(q_0', s)$ setzen. Im folgenden zeigen wir, dass diese Funktion q_0 auf q_0' und F auf F' abbildet und im gesamten einen Isomorphismus darstellt.

Ersteres ist einfach zu sehen:

$$\begin{aligned} \phi(q_0) &= \phi(row(\lambda)) \\ &= \delta'(q_0', \lambda) \\ &= q_0' \end{aligned}$$

Im Folgenden zeigen wir, dass Kanten aus $M(S, E, T)$ ebenfalls in M' existieren. Sei s aus S, a aus Σ und s_1 aus S, so dass $row(s \cdot a) = row(s_1)$ ist. Dann folgt:

$$\begin{aligned} \phi(\delta(row(s), a)) &= \phi(row(s \cdot a)) \\ &= \phi(row(s_1)) \\ &= \delta'(q_0', s_1) \end{aligned}$$

Ebenfalls gilt:

$$\begin{aligned} \delta'(\phi((row(s)), a) &= \delta'(\delta'(q_0', s), a) \\ &= \delta'(q_0', s \cdot a) \end{aligned}$$

Nun müssen $\delta'(q_0', s_1)$ und $\delta'(q_0', s \cdot a)$ den gleichen Zustand bezeichen, da $row(s_1)$ und $row(s \cdot a)$ den gleichen Wert haben. Wir schließen daraus dass $\phi(\delta(row(s), a)) = \delta'(\phi(row(s)), a)$ für alle $s \in S, a \in Sigma$ gilt. Dass ϕ F auf F' abbildet ist einfach einzusehen, da beide Automaten konsistent zu T sind.

Wir haben also gezeigt, dass $M(S, E, T)$ der **kleinste und einzigartige Automat konsistent** zu T ist.

3.2 Learner

In dem $\mathcal{L}^*$-Algorithmus wird eine anfangs leere OT ($S = E = \{\lambda\}$) von dem so genannten *Learner* verwaltet. Die Funktion T wird mit $a \in \Sigma : T(a) = Membership(a)$ initialisiert. Der eigentliche Kern des Algorithmus steckt in den zwei geschachtelten Schleifen. In der inneren Schleife wird die aktuelle OT (S, E, T) darauf getestet, ob sie *closed* und *consistent* ist.

Falls (S, E, T) nicht *consistent* ist, wähle $s_1, s_2 \in S$, $a \in \Sigma$, und $e \in E$, so dass $row(s_1) = row(s_2)$ und $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$, füge $a \cdot e$ zu E hinzu und ergänze T durch entsprechende $Membership$-Anfragen. Es ist leicht nachzuvollziehen, dass $E \cup \{a \cdot e\}$ ebenfalls unter Suffix abgeschlossen ist.

Sollte (S, E, T) nicht *closed* sein, dann finde ein $s_1 \in S$ und $a \in \Sigma$, so dass $row(s_1 \cdot a) \neq row(s) \forall s \in S$, füge $s_1 \cdot a$ zu S hinzu und ergänze T durch $Membership$-Anfragen. S bleibt hierbei und Präfix abgeschlossen.

Diese zwei Operationen werden in der inneren Schleife wiederholt, bis die OT *closed* und *consistent* ist. Sobald dies der Fall ist, wird in der äusseren Schleife eine $Conjecture$-Anfrage mit $M(S, E, T)$ an den $Teacher$ gestellt. Dieser antwortet entweder mit JA, dann kann mit der Ausgabe von $M(S, E, T)$ der Algorithmus beendet werden, oder er antwortet mit einem Gegenbeispiel t. In diesem Fall wird t und alle seine Suffixe zu E hinzugefügt und T durch $Membership$-Anfragen ergänzt. Der Algorithmus beginnt dann wieder von vorne, unter Berücksichtigung der erweiterten OT aus dem letzten Durchgang. Hier eine des $\mathcal{L}^*$ in Pseudocode:

Initialisiere $S = \lambda$ und $E = \lambda$.
Erzeuge initiale OT mit $T(\lambda) = Memberhsip(\lambda)$
und initialisiere T mit $\forall a \in \Sigma : T(a) = Membership(a)$.
Repeat:
    While (S, E, T) ist nicht *closed* und *consistent*
        If (S, E, T) ist nicht *consistent*
          finde $s_1, s_2 \in S$, $a \in \Sigma$, und $e \in E$ so dass
          $row(s_1) = row(s_2)$ und $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$,
          füge $a \cdot e$ zu E hinzu
          und ergänze T durch $Membership$-Anfragen.
        If (S, E, T) ist nicht *closed*
          finde $s_1 \in S$ und $a \in \Sigma$ so dass
          $row(s_1 \cdot a) \neq row(s) \forall s \in S$
          füge $s_1 \cdot a$ zu S hinzu
          und ergänze T durch $Membership$-Anfragen.
    Sobald (S, E, T) *closed* und *consistent* ist, erzeuge den Automat $M = M(S, E, T)$.
    Übergebe M der $Conjectur$-Anfrage
    If der $Teacher$ antwortet mit einem Gegenbeispiel t
        füge t und all seine Präfixe zu S hinzu
        und ergänze T durch $Membership$-Anfragen.
Until der $Teacher$ antwortet mit JA auf die $Conjectur$-Anfrage.
Ausgabe von M.

Der *Learner* versucht also, beginnend mit einer OT $S = \{\lambda\}$,$E = \Sigma$ einen ersten minimalen Automaten zu erzeugen, der mit T konsistent ist. In dem *"'If* (S, E, T) *ist nicht consistent"'*-Block werden ein Zustände $row(s_1) = row(s_2)$ identifiziert, welcher aufgrund der Kante a und dem Suffix e in mindestens zwei Zustände aufgeteilt werden muss, dies geschieht wiederum durch die Aufnahme von $a \cdot e$ zu E. In dem *"'If* (S, E, T) *ist nicht closed"'*-Block werden bildlich gesprochen Kanten gesucht, die zu keinem Zustand zeigen. Das heißt, es wird ein Zustand $row(s)$ und ein $a \in \Sigma$ gesucht, so dass der Eintrag $row(s \cdot a)$ bisher noch nicht existiert. Dieser Zustand wird dann durch die Aufnahme von $s \cdot a$ in S hinzugefügt.

Sobald Konsistenz und Geschlossenheit der OT gewährleistet ist, wird der entsprechende Automat konstruiert und als Vorschlag an den $Teacher$ übergeben. Dieser antwortet entweder mit JA und wir sind fertig oder aber mit einem Gegenbeispiel, welches dann unter Berücksichtigung all seiner Präfixe zu S hinzugefügt wird und die Hauptschleife von vorne startet.

Die *Korrektheit* des $\mathcal{L}^*$-Algorithmus ist leicht einzusehen, da der Algorithmus nur terminiert, wenn der *minimale adequate Teacher* die *Conjectur*-Anfrage positive beantwortet.

Zur Terminierung des $\mathcal{L}^*$-Algorithmus benötigen wir folgendes Lemma:

Lemma 4. *Sei $M(S, E, T)$ eine OT und bezeichne m die Anzahl an verschiedenen Werten von $row(s)$ für s aus S, dann hat jeder mit T konsistente Automat mindestens m Zustände.*

Beweis : Sei $M = (\mathcal{Q}, \delta, q_0, \mathcal{F})$ ein mit T konsistenter Automat. Wähle s_1, s_2 aud S, mit $row(s_1) \neq row(s_2)$. Das heißt, es existiert mindestens ein $e \in E$ mit $T(s_1 \cdot e) \neq T(s_2 \cdot e)$. Für den Automat M bedeutend dies aber, dass nur einer der Zustände $\delta(q_0, s_1 \cdot e), \delta(q_0, s_2 \cdot e)$ in $\mathcal{F}$ sein kann. Daher müssen $\delta(q_0, s_1), \delta(q_0, s_2)$ verschiedene Zustände sein und da es m verschiedene Werte für $row(s)$ gibt, muss auch der Automat M mindestens m verschiedene Zustände haben. □
Nehmen wir nun an, dass n die Anzahl der Zustände in dem minimalen Automat M_U ist, welcher U erkennt. Wir zeigen im Folgenden, dass die Anzahl der verschiedenen Werte für $row(s)$ während dem Ablauf von $\mathcal{L}^*$-Algorithmus monoton anwächst und n als obere Schranke hat.

Angenommen ein String $(a \cdot e)$ wird zu E hinzugefügt, weil die OT nicht *consistent* war. Dann muss sich die Anzahl der unterschiedlichen Einträge für $row(s)$ um mindestens einen erhöhen, da es vorher zwei inkonsistente Werte $row(s_1) = row(s_2)$ existierten, die nun durch das Hinzufügen $(a \cdot e)$ unterschieden werden können. Des weiteren ist klar, dass zwei vorher unterschiedliche Werte $row(s_3) \neq row(s_4)$ durch das Erweitern von E weiterhin unterschiedlich bleiben.

Angenommen ein String $s_1 \cdot a$ wird zu S hinzugefügt, weil die OT nicht *closed* ist. Dann erhöht sich die Anzahl der verschiedenen Einträge für $row(s)$ in S genau um den hinzugefügten $row(s_1 \cdot a$, da dieser vorher noch nicht in S enthalten gewesen sein konnte, dies würde im Widerspruch zur Definition von *closed* stehen.

Der $\mathcal{L}^*$-Algorithmus -Algorithmus kann also insgesamt maximal $n-1$ oben genannter Operationen durchführen. Zu betrachten bleibt die *conjecture*-Anfrage. Sei t das Gegenbeispiel, welches von dem $Teacher$ auf den Vorschlag von $M(S, E, T)$ zurückgegeben wurde. Da $M(S, E, T)$ *closed*, *consistent* und *minimal* war, dies aber nicht mehr bezüglich des neuen Beispiels t ist, muss der Automat M', welcher aus der um t erweiterten OT hervor geht, nach Theorem 1 mindestens einen weiteren Zustand besitzen. Zusammengefasst sieht man also, dass die Hauptschleife und die innere Schleife zusammen maximal $n-1$ mal ausgeführt werden können ($n-1$ mal, da die OT anfangs aus einem Zustand besteht).

Damit wurde die Korrektheit und die Terminierung des $\mathcal{L}^*$-Algorithmus -Algorithmus gezeigt.

4 Der $\mathcal{L}^\omega$-Algorithmus

Wir wollen in diesem Kapitel das Lernen von $\mathcal{B} \cap \bar{\mathcal{B}}$ -Sprachen betrachten und dabei den $\mathcal{L}^*$ zu dem $\mathcal{L}^\omega$-Algorithmus ausbauen. Zuerst müssen wir die Definition der Observation Table anpassen, so dass diese in Zukunft für die Aufnahme von ultimativ periodischen Wörtern geeignet ist. Des weiteren bezeichnen wir die zu lernende Sprache mit $\mathcal{U}$. Die Methoden $Member$ und $Equiv$ des $Teachers$ liegen hier selbstverständlich als "'Omega"'-Varianten vor. Das bedeutet $Equiv$ liefert ein ultimativ periodisches Wort im Fehlerfall zurück und $Member$ bekommt eines als Parameter übergeben.

4.1 Der $\mathcal{L}^\omega$-Algorithmus im Überblick

Im Folgenden eine Zusammenstellung des $\mathcal{L}^\omega$-Algorithmus in Pseudocode:

```
Initialisiere S = {λ} und E = {a^ω : a ∈ Σ}.
Erzeuge initiale OT und initialisiere T für die Werte
aus ((S ∪ S · Σ) · E) durch Membership-Anfragen.

While !confirmed do
        While the OT ist not closed do
                füge den Zustand sa zu S hinzu
                und erweitere die OT entsprechend
        end loop
        Konstruiere den Automat aus der OT
        Rufe Marking-Methode auf
        If Marking findet ein Conflict then
                Rufe ConflictResolution auf
                und füge das Resultat und seine Suffixe zu E hinzu
        else
                Rufe Equiv für den aktuell markierten Automaten auf
                If Equiv antwortet JA then
                        confirmed = true
                else
                        füge das Gegenbeispiel von Equiv und seine Suffixe zu E hinzu.
                end if
        end if
end loop
```

4.2 ω-Observation Table

Definition 8. ***ω-Observation Tables****: Eine $\omega - OT$ ist ein Tripel (S, E, T). S ist eine unter Suffix abgeschlossene Teilmenge von Σ^*, E ist eine unter Präfix abgeschlossene Menge von ultimativ periodischen Wörtern. T stellt eine Abbildung von $(S \cup S\Sigma) \times E \to \{0,1\}$ dar, wobei für $s \in \Sigma^\omega$ stets $T(s) = 1 \Leftrightarrow s \in \mathcal{U}$ gilt. Wir führen für jedes $s \in S$ die Funktion $f_s : E \to \{0,1\}$ ein, die durch $f_s(\alpha) = T(s \cdot \alpha)$ definiert ist.*

Die Definition von ***closed*** ändert sich durch das arbeiten mit unendlichen Wörtern nicht. Auch muss die Eigenschaft ***consistent*** nicht mehr überprüft werden, da im Gegensatz zu $\mathcal{L}^*$ nun die Gegenbeispiel und ihre Suffixe zu E hinzugefügt werden und dadurch die Konsistenz der OT automatisch garantiert ist.

Definition 9. *Jeder closed $\omega-OT$ lässt sich in folgender Weise ein* ***Transitionsgraph***$(\Sigma, \mathcal{Q}, \delta, q_0)$ *zuordnen:*

$$\begin{aligned} \mathcal{Q} &= \{f_s : s \in S\} \\ q_0 &= f_\lambda \\ \delta(f_s, a) &= f_t, t \in S \wedge f_{sa} = f_t \end{aligned}$$

Wie man sieht, sind hier die Endzustände $\mathcal{F}$ nicht definiert worden. Im endlichen Fall konnten Endzustände leicht durch $f_s(\lambda) = 1$ identifiziert werden. In unserem unendlichen Fall aber, ist dies nicht so einfach möglich. Deshalb musste auch die Sprachklasse auf $\mathcal{B} \cap \bar{\mathcal{B}}$ -Sprachen eingeschränkt werden, denn für solche Sprachen lassen sich die Endzustände wie folgt identifizieren.

Definition 10. *Für eine ω-OT ist ein* ***Conflict*** *wie folgt definiert:*
Ein $Conflict$ ist ein Tripel (s, u, v), so dass $f_{su} = f_{sv} = f_s$ gilt, aber $f_s(u^\omega) \neq f_s(v^\omega)$

Ein $Conflict$ gibt also an, dass ein Zustand f_s existiert, der zu einer MSCC gehört, die nicht einheitlich aus Endzuständen, bzw. Nicht-Endzuständen besteht. Ein solcher $Conflict$ wird später verwendet um f_{su} und f_{sv} zu unterscheiden und so sukzessive alle MSCC's zu vereinheitlichen.

4.3 Die *Marking*-Methode

Da laut Satz 1 ein Automat mit einheitlicher Endzustandsmarkierung für alle MSCC's für die Sprache $\mathcal{U}$ existiert, überprüft die $Marking$-Methode, ob für den aus der $\omega - OT$ gewonnene Transitionsgraph konfliktfrei eine solche Markierung der Zustände existiert und gibt im Fall des Erfolgs den fertigen Automaten zurück, ansonsten wird eine entsprechender $Conflict$ zurück gegeben.

Die Marking-Methode arbeitet dabei in vier Stufen:

1. Es werden alle Zustände des Graphen als "'unbehandelt"' markiert.

2. Bestimme für jedes ω-Wort $\alpha = s \cdot e, s \in S, e \in E$ die Zustände $Inf(\alpha)$ und markiere diese mit $T(\alpha)$. Sollte dabei versucht werden, ein Zustand f_z mit zwei unterschiedlichen Werten zu markieren, wurde ein Konflikt gefunden. Bestimme die Zyklen u, v die für die positiv, bzw. negativ Kategorisierung verantwortlich sind und gib den $Conflict$ (z, u, v) zurück.

3. Berechne die MSCC des Graphen. Markiere die MSCC's einheitlich, falls dies möglich ist und gebe diesen zurück.

4. Falls dies nicht möglich war, bedeutet dies, dass innerhalb einer MSCC zwei Zustände f_s, f_t existieren so dass sx^ω ein positives und ty^ω ein negativ Beispiel ist. Um nun einen $Conflict$ zu kreieren, benötigen wir einen Zustand von dem aus ein akzeptierender und ein zurückweisender Zyklus existiert. Hierzu untersuchen wir den Zyklus zw, wobei z von f_s nach f_t und w von $f_t$nach f_s führt. Dieser muss existieren, da wir uns in einer MSCC befinden. Stelle eine Member-Anfrage mit $s(zw)^\omega$. Im Falle einer positiven Antwort, gebe (t, y, wz) zurück, ansonsten (s, x, zw).

Sollte die $Marking$-Methode erfolgreich ablaufen, wird der resultierende Automat dem $Teacher$ per $Conjecture$-Anfrage übergeben und abhängig von dessen Antwort der erfolgreich gelernte Automat ausgegeben oder das Gegenbeispiel mit allen seinen Suffixen zu E hinzugefügt und die Hauptschleife von vorne begonnen.

Wird andernfalls ein $Conflict$ entdeckt muss dieser entsprechend behandelt werden, um daraus ein ω-Wort zu konstruieren, welches dann zu E hinzugefügt wird und die im Konflikt zueinander stehenden Zyklen differenziert. Dies betrachten wir im folgenden Abschnitt.

4.4 Die *ConflictResolution*-Methode

Definition 11. *Wir definieren $\Delta^n_{u,v}$ für $u, v \in \Sigma^+$ als Menge von Wörtern der Form xu^ω oder xv^ω, wobei x ein Präfix von $(u^n v^n)^n$ oder $(v^n u^n)^n$ ist.*

Satz 2. *Sei $\mathcal{A} = (\Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F})$ ein $\mathcal{B} \cap \bar{\mathcal{B}}$ - Automat, mit $\mathbb{L}_A = \mathcal{U}$ und sei $su^\omega \in \mathcal{U}$ und $sv^\omega \notin \mathcal{U}$. Offensichtlich muss mindestens einer der Zustände $\{\delta(q_0, su), \delta(q_0, sv)\}$ von $\delta(q_0, s)$ verschieden sein. Sei $t \in \{su, sv\}$ ein endliches Wort, so dass $\delta(q_0, t) \neq \delta(q_o, s)$ gilt. Dann existiert ein Wort $\alpha \in \Delta^n_{u,v}$, welches t und s unterscheidet und somit diesen $Conflict$ behebt.*

Beweis: O.B.d.A. nehmen wir an, dass $t = sv\wedge \Rightarrow \delta(q_0, s) \neq \delta(q_0, t)$. Das Wort sv^n führt offensichtlich zu einer zurückweisenden MSCC, welche von sv^ω unendlich oft besucht wird. Es gibt zwei Möglichkeiten für svu^ω:

1. $sv^n u^\omega$ ist zurückweisend, in diesem Fall gibt es ein $k, 0 \leq k < n$, so dass $sv^k u^\omega \in \mathcal{U}$ und $sv^{k+1}u^\omega \notin \mathcal{U}$ und sich daraus ergibt, dass $\alpha = v^k u^\omega$ zwischen s und sv unterscheidet.

2. Wenn hingegen $sv^n u^\omega$ akzeptiert wird, dann führt $sv^n u^n$ zu einem neuen akzeptierenden MSCC. Da die Anzahl der MSCC's durch n beschränkt ist, ist auch die Anzahl der Alternierungen zwischen akzeptierenden und zurückweisenden MSCC's durch n beschränkt. Deshalb existiert ein unterscheidendes $\alpha = v^n u^n \dots u^n v^k u^\omega$ oder $\alpha = v^n u^n \dots v^n u^k v^\omega$

Die ***ConflictResolution***-Methode macht sich dies zu Nutze, indem sie $\Delta^n_{u,v}$ für fortlaufend erhöhendes n aufzählt und dabei $Membership$-Anfragen für $s\alpha, sv\alpha$ und $su\alpha$, solange bis $Member(s\alpha) \neq Member(sv\alpha)$ oder $Member(s\alpha) \neq Member(su\alpha)$ eintritt. Dann wird $v\alpha$ bzw. $u\alpha$ zurück gegeben, dementsprechend welche Ungleichheit eingetroffen ist.

5 Die Implementierung

In diesem Kapitel wollen wir die für diese Studienarbeit ausgefertigte Implementierung des $\mathcal{L}^{\omega}$-Algorithmus betrachten.

Da wir uns nun im Folgenden über Ω -Wörter unterhalten, müssen wir uns Gedanken über die interne Repräsentierung machen. Wie schon erwähnt, sind wir nur auf die *ultimativ periodischen*-Wörter angewiesen, diese lassen sich leicht in zwei *String*-Variablen speichern. In der Implementierung wird der Klasse $Word$ neben den zwei Strings noch eine Integer-Variable hinzugefügt, diese dient aber lediglich zu Formatierungszwecken.

Die ***Observation Table*** wird als zwei einfach verkettete Kammlisten implementiert. Zwei Kammlisten, da die Präfixe S und $(S \cup \Sigma)$ getrennt verwaltet werden. Die Erste Spalte in einer solchen Kammliste dient der Beschriftung mit den endlichen Präfixe, die erste Zeile der Beschriftung durch die Suffixe aus E. Die Einträge entsprechen dem Wert der Funktion T, können also nur Werte aus $\{0, 1\}$ annehmen.

Zuerst betrachten wir die Umsetzung des $Teachers$, wir benötigen dies, um die Implementierung testen zu können. In dem Paper [Pn/Ma] wird von zwei Orakeln ausgegangen, $Membership$ und $Equiv$, die entsprechende Anfragen beantworten können. Da uns diese Orakel leider nicht zur Verfügung stehen, müssen wir uns anderweitig behelfen. Wir verwenden diesbezüglich einen Automaten, um oben genannte Funktionalität zu realisieren.

Das zugrunde liegende ***Automaten-Modell***: Die Automaten (Teacher und Learner) werden als Adjazenzliste gespeichert. Des weiteren wird den Knoten einiges an Zusatzinformation angeheftet, dazu aber mehr an entsprechenden Stellen. Die Knoten haben beispielsweise die Möglichkeit als Endzustand markiert. Es wurde der Einfachheit wegen immer der Erste Knoten aus der Adjazenzliste als Startknoten definiert. Weiterhin sind jedem Knoten zwei Listen (Index1, Index2) angehängt die wir für folgende Methode benötigen.

Die ***Membership***-Methode: Um festzustellen ob ein gegebenes *ultimativ periodisches* Wort von einem Automat erkannt wird, muss überprüft werden, ob diesen, wenn es von dem Automat gelesen wird, unendlich oft mindestens einen Endzustand besucht. Zuerst wird der endliche Präfixe von dem Automat gelesen. Dann wird der sich wiederholende Rest gelesen, hierbei wird für jeden Knoten der Index des Zeichens festgehalten, welches gelesen wurde, um diesen zu besuchen. Sobald wir einen Index das zweite Mal an einem Knoten betrachten, haben wir einen Zyklus durchlaufen und alle dazugehörenden Knoten werden von diesem Wort unendlich oft besucht. Es bleibt nun lediglich übrig, den Zyklus ein weiteres Mal zu durchlaufen, dabei zu überprüfen ob mindestens ein Endzustand besucht wird und das entsprechende Resultat zurückzugeben.

Die ***Equiv***-Methode: Um zwei Büchi-Automaten auf Äquivalenz zu testen, können wir uns hier auf die Äquivalenz bezüglich *ultimativ periodischer* Wörter beschränken. Hierzu wird ein Kreuzautomat aus beiden Automaten aufgestellt. Wichtig ist zu beachten, dass in diesem Kreuzautomaten jeder Zustand die Möglichkeit hat, kein Endzustand zu sein, nur von einem der beiden Automaten ein Endzustand zu sein oder aber von beiden Automaten. Sollte bei der Konstruktion der Fall auftreten, dass von einem Knoten aus, nur für einen von beiden Automaten eine Kante mit der Beschriftung γ vorkommt, kann sofort abgebrochen werden, insofern dieser vom Startzustand aus erreichbar ist (Erreichbarkeit per Breitensuche). Als Gegenbeispiel wird nun $z \cdot \gamma^\omega$ zurückgeliefert, wobei z ein Pfad vom Startzustand zu dem entsprechenden Konfliktzustand darstellt, welches leicht durch Breitensuche ermittelt werden kann.

Es werden nun von allen erreichbaren Knoten im Kreuzautomat alle möglichen einfachen Zyklen daraufhin überprüft, ob sie entweder Endzustände aus beiden beziehungsweise keinem der zugrunde liegenden Automaten enthalten. Sollte ein Zyklus gefunden werden, der dem widerspricht, kann wieder ein Gegenbeispiel $z \cdot \gamma^\omega$ geliefert werden. Hierbei bezeichnet z einen Pfad zum Anfang des Zyklus und γ den Zyklus selbst. Um alle Zyklen untersuchen zu können, wird für alle erreichbaren Knoten k ***Cycle_ Check*** aufgerufen. *Cycle_ Check* sucht mittels Tiefensuche alle Zyklen die in k beginnen und enden. Dabei wird ein Array verwaltet, welches Referenzen auf alle derzeit abgelaufenen Knoten enthält. Sobald nun ein Zyklus identifiziert wurde, kann mittels des Arrays überprüft werden, ob Endzustände aus keinem, beiden oder nur einem von beiden Automaten besucht wurden. Dabei wurde die Tiefensuche so ausgelegt, dass für den Widerspruchsfall gleich ein entsprechender String mitgeführt wird, welcher den entsprechenden Zyklus durchläuft. Dieser wird dann zurückgegeben und von *Equiv* erkannt. *Equiv* sucht nun lediglich den endlichen Präfix in bekannter Weise und setzt daraus das Gegenbeispiel zusammen.

Sollte kein widersprechender Zyklus gefunden werden, wird selbstverständlich JA zurückgegeben, um anzuzeigen, dass beide Automaten äquivalent sind.

Im Folgenden werden die wichtigsten Methoden des *Learner* vorgestellt. Dabei wird die Reihenfolge aus dem Algorithmus beibehalten, um so auch dessen Ablauf mit erklären zu können:

Die ***NotClosed***-Methode: Überprüft ob es für jeden Eintrag $row(s \cdot a)$in der Tabelle für $S \cup \Sigma$ einen passenden Eintrag in der Tabelle für S existiert. Sobald ein Eintrag $row(s \cdot a)$ gefunden wird, welcher noch nicht in S existiert, wird dessen Zeilennummer zurückgegeben. Sollte die OT schon *closed* gewesen sein, wird lediglich 0 zurückgegeben werden. Hier weicht die Implementierung von der vorlage aus [Pn/Ma] ein wenig ab, da die Abfrage *not closed* in die folgenden Methode mit eingebaut wurde.

Die ***CloseTable***-Methode: In dieser Methode wird sukzessive $NotClosed$ aufgerufen und die zurückgegebene Zeilennummer dazu verwendet, die konflikt auslösenden Zustände aus $S \cup \Sigma$ nach S zu verschieben. Hierfür wird eine weiter Methode $Unlink$ verwendet, die einen Eintrag aus der einen Liste in die andere einfügt. Des weiteren

muss für jeden verschobenen Eintrag $row(s \cdot a)$ für jedes $b \in \Sigma$ dann noch $s \cdot a \cdot b$ zu $S \cup \Sigma$ hinzugefügt werden.

Die ***UpdateTable***-Methode: Da die OT ständig um weitere Einträge erweitert wird, müssen auch die daraus resultierenden neuen Werte für T gesetzt werden. $UpdateTable$ läuft hierzu alle Einträge der OT ab und ruft für jeden nicht gesetzten Wert für T die Membership-Methode auf und setzt den Wert entsprechend dem Resultat.

Die ***Observation Table to Automaton***-Methode: Sobald eine *closed* OT vorliegt, muss für diese ein Automat erzeugt werden. Zuerst wird eine Zustands-Liste mit den Werten $(s, row(s))$ aus S erzeugt, dabei werden mehrfach vorkommende Werte von $row(s)$ nur einmal betrachtet, da diese den selben Zustand bezeichnen. Danach wird analog dazu eine Kanten-Liste mit den Werten $(s \cdot a, row(s \cdot a))$ aus $S \cup \Sigma$ erstellt. Es wird zunächst ein Automat erzeugt, welche für jeden Eintrag aus der Zustands-Liste einen Zustand erhält. Der Erste Zustand entspricht dem Startzustand und gleichzeitig dem Eintrag $(\lambda, row(\lambda)$. Alle Zustände entsprechen mit ihrer Position in der Adjazenzliste auch der Position aus der Zustands-Liste, so lassen sich im Folgenden leicht die fehlenden Kanten einfügen.

Für das Einfügen der Kanten, benötigen wir beide vorher erzeugte Listen. Für beide Listen wird für jeden Einträge $(t \cdot a, row(t \cdot a)), a \in \Sigma$ in der Zustands-Liste zuerst der Eintrag $(t, row(t))$ und dann der Eintrag $(t \cdot a, row(t \cdot a))$ gesucht und jeweils der dazugehörige Index gespeichert. Es bleibt nur noch in dem Automaten eine Kante von dem Zustand mit dem ersten Index zu dem Zustand mit dem zweiten Index zu erzeugen und diese mit a zu beschriften.

In der Implementierung wird für den so erzeugten Automat die *Complete*-Methode aufgerufen, diese überprüft zunächst, ob der Automat vollständig ist. Fall dies der Fall ist belässt sie den Automat wie er ist. Falls mindestens eine Kante fehlt, fügt sie einen weiteren Zustand hinzu und lässt alle nicht vorhandenen Kanten auf diesen verweisen, insbesondere die Kanten von diesem neuen Knoten aus.

Die ***Marking***-Methode: Wir versuchen in dem so gewonnen Automaten die Endzustände zu identifizieren und zu markieren. Wie in Kapitel 4 beschrieben, kann es hierbei ein Konflikt-Fall auftreten. In diesem Fall wird der $Conflict$ zurückgegeben. Zuerst werden für alle Wörter s aus der OT die Zustände des Automaten ermittelt, die bei lesen dieses Wortes unendlich oft besucht werden und dann mit $(s, T(s))$ markiert. Hierbei kann der Fall auftreten, dass ein Zustand z der bereits mit $(s, T(s))$ markiert ist, nun mit $(r, T(r)), T(r) \neq T(s)$ markiert werden soll. Dies ist der erste Konflikt-Fall. Der $Conflict$ (s, u, v) setzt sich wie folgt zusammen: s ist ein Pfad vom Startzustand zu z, u ein Pfad startend und endend bei z, welcher dabei dem Zyklus von r folgt, v wiederum folgt hierbei dem Zyklus von s.

Insofern das bisherige Markieren der Zustände ohne Konflikte erfolgte, werden nun für den weiteren Verlauf des Algorithmuses die ***maximalen starken Zusammenhangskomponenten (MSCC)*** bestimmt. Hierzu eignet sich der Algorithmus von Tarjan, auf

den hier nicht näher eingegangen wird. Zu überprüfen bleibt nun lediglich, ob alle Zustände einer MSCC einheitlich als *akzeptierend* beziehungsweise *zurückweisend* markiert wurden. Sollten zwei Zustände einer MSCC mit unterschiedlicher Markierung gefunden werden, muss hieraus ein $Conflict$ konstruiert werden. Sei o.B.d.A der erste Zustand $\hat{s}$ akzeptierend und der zweite Zustand $\hat{t}$ zurückweisend und bezeichnen s, t die dazugehörigen Pfade vom Startzustand zu dem jeweiligen Zustand. Da sich beide Zustände im selben MSCC befinden, muss es ein Pfad z von $\hat{s}$ nach $\hat{t}$ und einen Pfad w von $\hat{t}$ nach $\hat{s}$ geben, welche sich durch Breitensuche bestimmen lassen. Des weiteren gibt es den Zyklus x, welcher von $\hat{s}$ aus startet und zu dem akzeptierten Wort gehört. Analog dazu gibt es den Zyklus y, in $\hat{t}$ startend, zu dem zurückgewiesenen Wort gehörend. Da ein Knoten gesucht wird, von dem aus zwei widersprüchliche Zyklen aus beginnen, wird als nächstes der Zyklus $z \cdot w$ in $\hat{s}$ startend betrachtet. Es wird eine Membership-Anfrage mit $(s, (z \cdot w)^\omega)$ gestellt. Ist die Antwort Positiv, setzt sich der $Conflict$ aus $(t, y, w \cdot z)$ zusammen, andernfalls aus $(s, x, z \cdot w)$.

Zu betrachten bleibt der Fall, dass keine Konflikte auftreten. Dann können alle Zustände einzeln betrachtet werden und jeder Zustand, der Anfangs für ein Wort aus der OT als akzeptierend markiert wurde, kann schließlich als Endzustand markiert werden. Es gilt lediglich zu beachten, dass sollte ein Zustand beim Vervollständigen des Automaten hinzugekommen sein, dieser natürlich nicht als Endzustand dienen darf.

Die ***ConflictResolution***-Methode: Diese Methode stellt sicherlich das Herzstück des Papers [Pn/Ma] und auch des $\mathcal{L}^\omega$-Algorithmus dar. Übergeben bekommt diese Methode einen $Conflict$ (s, u, v), aus dem sie dann unter Zuhilfenahme von Membership-Anfragen ein Omega-Wort zusammengesetzt, welches die den Konflikt verursachende Zyklen separiert. Die im Paper eingeführte Menge $\Delta^n_{u,v} = \{w | w = xu^\omega \vee xv^\omega, x \in \text{praefix}((v^n u^n)^n | (u^n v^n)^n)\}$ wird im Gegensatz zu dem Vorschlag aus dem Paper [Pn/Ma] nicht der Länge nach sortiert durchprobiert, sondern es wird abwechselnd und blockweise $(u^n v^n)^n$ beziehungsweise $(v^n u^n)^n$ als Präfix verwendet. Mit jedem Schleifendurchgang wird dabei n um eins hoch iteriert. Sei $\alpha \in \Delta^n_{u,v}$ das derzeit betrachtete Element, dann werden die Wörter $s\alpha, sv\alpha$ und $su\alpha$ zusammengesetzt und jeweils Membership-Anfragen gestellt, solange bis entweder $T(s\alpha) \neq T(sv\alpha)$ oder $T(s\alpha) \neq T(su\alpha)$ gilt. Dementsprechend welcher Fall eintritt, wird $u\alpha$ bzw. $v\alpha$ zurückgegeben.

Egal ob aus $Equiv$ oder $ConflictResolution$ ein Omega-Wort zurückgegeben wird, muss dieses und alle seine Suffixe zu E hinzugefügt werden. Für diesen Zweck wurde die Methode $SuffixAdding$ eingeführt. Diese erzeugt aus einem ***ultimativ periodischen*** Wort $\alpha\beta^\omega : \alpha, \beta \in \Sigma^*$ alle Suffixe, in dem zuerst jeweils das Anfangszeichen von α gelöscht wird, bis dieses ganz weggefallen ist. Da die Suffix-Relation keine Ordnung auf unendlichen Wörtern darstellt, muss der *beta*-Teil noch zeichenweise durch rotiert werden, um alle Suffixe zu erzeugen. Hierfür muss aber lediglich sukzessiv das erste Zeichen hinten angefügt werden, solange bis man wieder β erhält.

Der komplette Ablauf des $\mathcal{L}^{\omega}$-Algorithmus im Überblick:

1. Die OT solange erweitern, bis sie *closed* ist
2. Einen Markierungsversuch mit der OT starten
3. Wenn beim Markieren ein $Conflict$ aufgetreten ist, dann weiter bei 4. ansonsten bei 5.
4. $ConflictResolution$ aufrufen und das Resultat und seine Suffixe zu E hinzufügen. Dann zurück zu 1.
5. Den Automat an $Equiv$ übergeben
6. Falls diese mit einem Gegenbeispiel antwortet, dieses und seine Suffixe zu E hinzufügen.
7. Solange zu ***1.*** zurückspringen, bis $EQUIV$-Anfrage positiv ausfällt

6 Adaptive Model Checking

Das hier vorgestellte Verfahren eignet sich besonders wenn bereits ein Model existiert, welches das System nur teilweise beschreibt oder noch Fehler enthält. Es kann dazu verwendet werden, ein unvollkommenes Modell teilweise automatisiert zu ergänzen und dafür eingesetzt werden Fehler im Modell und System aufzuspüren. Wenn das Modell kleinere Fehler enthält können diese automatisch korrigiert werden. Das Verfahren ähnelt in seinem Grundprinzip stark dem Black-Box-Checking, bei welchem von Grund auf ein neues Modell aufgebaut wird. Bei dem Adaptive-Model-Checking Ansatz wird die grundsätzliche Vorgehensweise des Black-Box-Checking übernommen und dahingehen erweitert, bereits vorhandene Informationen bezüglich des Systems und über das Model zu verwenden, um möglichst viele Aufrufe des laufzeitintensiven Vasilvskii-Chow-Algorithmus einzusparen.

Der AMC-Ansatz kann für folgende Fälle besonders effizient eingesetzt werden:

- wenn das Modell Modellierungsfehler enthält.
- nachdem ein Fehler in dem System behoben wurde.
- wenn das System um ein Merkmal erweitert wurde.
- wenn eine neue Version des Systems erscheint.

Die erste Phase des AMC stellt das Lernverfahren von Angluin dar. Diese wurde schon in Kapitel 3 beschrieben. Es werden kleinere Anpassungen vorgenommen, welche später beschrieben werden. In dieser Phase wird mit Hilfe eines bestehenden Model oder weiterer zuvor gewonnenen Informationen ein Model konstruiert, welches dann in der nächsten Phase für das Model-Checking zur Verfügung steht.

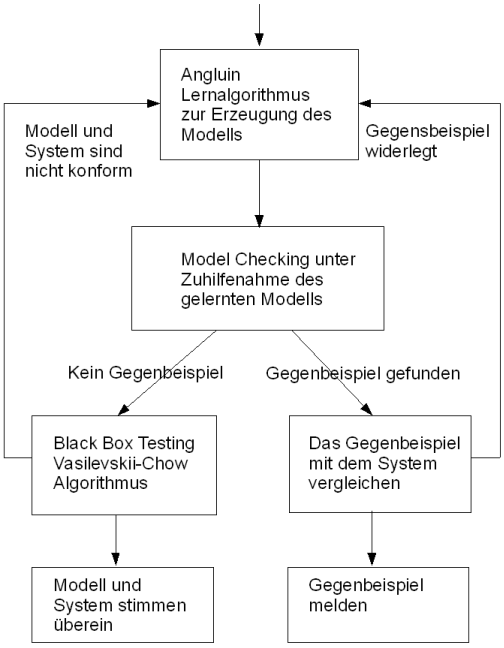
Für das Model-Checking können LTL-Formeln verwendet werden, welche wie gewohnt negiert und in einen Büchi-Automaten umgewandelt werden, um dann das Kreuzprodukt aus Model-Automaten und Formel-Automaten auf Leerheit zu überprüfen. Der so gewonnene Kreuzproduktautomat repräsentiert genau die Sprache der Gegenbeispiele. Es wird eine iterative Tiefensuche verwendet um möglichst kurze Gegenbeispiele zu erhalten, da die Länge der Gegenbeispiele maßgeblich für die Komplexität des Lernalgorithmuses verantwortlich ist.

Sollten bei dem Model-Checking keine Gegenbeispiele gefunden werden, muss das Modell mit dem System verglichen werden, da es noch nicht gewährleistet ist, dass dieses das System akkurat repräsentiert. Hierfür wird der sehr laufzeitintensive Vasilevskii-Chow Algorithmus verwendet, welcher in [T.Chow] beschrieben ist. Entweder bestätigt der VC-Algorithmus die Übereinstimmung von System und Model und es kann mir einer positiven Rückmeldung das Verfahren beendet werden. Andernfalls findet der VC-Algorithmus ein Gegenbeispiel, welches dann wieder dem Lernalgorithmus hinzugefügt wird um das Model entsprechend zu erweitern.

Sollte hingegen beim Model-Checking ein Gegenbeispiel gefunden werden, muss dieses zuerst mit dem System verglichen werden, um sicherzustellen, dass es sich hierbei auch

um eine gültige Ausführung des Systems handelt. Ist dies der Fall kann mit einer negativen Meldung und einem Report des Gegenbeispiels beendet werden, andernfalls wird auch hier wieder das Gegenbeispiel dem Lernalgorithmus hinzugefügt und dadurch das Model entsprechend angepasst.

Der gesamt Ablauf grafisch dargestellt:



Ein Model und ein System

Für die weiteren Betrachtungen ist es erforderlich die Begriffe System und Model zu konkretisieren. Ein Model ist wie gewohnt endlicher deterministischer Automat $M = (S, s_0, \Sigma, \delta)$, mit S ist die Menge der Zustände, s_0 der Startzustand, $Sigma$ das Alphabet und δ die Überführungsfunktion.
Ein Lauf von M sei wie gewohnt definiert. Die Sprache $\mathcal{L}(M)$ sei die Menge aller Läufe von M. Wir nennen einen Input a aktiv in einem Zustand s, wenn ein Zustand r existiert für den $(s, a, r) \in \delta$ gilt.

Ein System ist ein Zwei-Tupel (Σ, P), wobei Sigma das Alphabet und $P \subseteq \Sigma^*$ eine unter Präfix abgeschlossene (gewöhnlich unendliche) Menge ist. P beinhaltet alle möglichen Läufe des Systems. Es werden zwei Voraussetzungen für System benötigt. Es muss möglich sein, das System durch einen *Reset* in seinen Startzustand zurückzusetzen. Des weiteren muss es möglich sein, Experimente mit dem System durchzuführen. Ein Experiment ist eine Sequenz von Eingaben $e = a_1..a_n$ welche nach einem *Reset* sukzessive an das System übermittelt werden, welche diese ausführt. Dabei gibt das System Rückmeldung darüber, ob es möglich war, die Eingabe erfolgreich auszuführen. Ein Model

stimmt mit einem System genau dann überein, wenn alle erfolgreichen Experimente auch Läufe von M sind.

Lernalgorithmus von Angluin

Die Anpassung des Lernalgorithmus von [D.Angluin] wird benötigt, da es möglich sein soll, bereits vorhandene Informationen über die Observation Table (S, E, T)in das Programm mit einzubringen. Dies lässt sich leicht bewerkstelligen, indem dem Lernalgorithmus eine Parameterliste hinzugefügt wird, durch welche die Mengen S, T vorinitialisiert werden können oder ein Teil der Funktion T übergeben wird, so dass man sich das bestimmen schon bekannter Werte erspart.

Da der Lernalgorithmus verwendet wird, um ein Model zu lernen, dessen Sprache durch alle möglichen Läufe definiert wird, entfällt es, die Endzustände explizit anzugeben. Dafür werden für den konstruierten Automaten nur noch die Zustände $row(s)$ verwendet, für die $T(s) = 1$ gilt. Dies ist leicht einzusehen, mit oben genanntem Argument, dass ja alle Läufe zur Sprache des Model gehören und somit keine Zustände vorhanden sein dürfen, welche durch lesen eines Wortes erreicht werden können, welches kein gültiges Experiment des Systems darstellt.

Separierende Sequenzen

In Angluins Algorithmus wird die Menge E verwendet um die einzelnen Zustände zu unterscheiden. Hier interessant sind die separierenden Mengen, die für alle Zustände $s, t, s \neq t$ eine separiernde Sequenz enthalten. Mit Hilfe des Hopcroft Algorithmus lässt sich eine separierender Menge für einen gegebenen Automaten in $O(nlog(n)$ erzeugen.

Beschleunigung des Lernverfahrens

Die Grundvoraussetzung für das $AdaptiveModelChecking$ stellte ein Model, welches nicht fehlerfrei sein muss, aber zumindest einige nicht trivialen Eigenschaften des Systems teilt. Dieses Model lässt sich verwenden um einige Informationen zu gewinnen, welche das Lernverfahren stark beschleunigen können. Wenn eine zuvor erwähnte Separierende Menge für dieses Model berechnet wurde, ist die Wahrscheinlichkeit hoch, dass diese auch in dem korrekten Model viele Zustände separiert. Also muss lediglich eine Solche Menge mit dem Hopcroft-Algorithmus berechnet werden und als Startwert der Menge E für das Lernverfahren eingesetzt werden.

Zu solch einem gegebenen Model $M = (S, s_0, \Sigma, \delta)$ kann leicht ein Spannbaum konstruiert werden. Wenn man alle Läufe in solch einem Spannbaum betrachtet, erhält man dadurch eine Menge von Zugangssequenzen. Diese Zugangssequenzen eignen sich als Startwert der Menge S des Lernverfahrens, da die Wahrscheinlichkeit hoch ist, dass viele Zustände die im unvollkommenen Model über eine entsprechende Sequenz erreicht werden können, auch in dem korrekten Model sich über diese Sequenz erreichen lassen.

7 Abschlussbemerkung

Es wurden in dieser Arbeit die Algorithmen $\mathcal{L}^*$ und $\mathcal{L}^\omega$ vorgestellt, wobei der Zweite konzeptionell aus dem Ersten hervor geht. Für Σ^*-Sprachen wurde dabei ein allgemein gültiger Algorithmus präsentiert, wohingegen die Σ^ω-Sprachen auf die Klasse der $\mathcal{B} \cap \overline{\mathcal{B}}$ -Sprachen eingeschränkt werden mussten. In dem Kapitel 6 über Adaptive Modell Checking wurde der vorher vorgestellte $\mathcal{L}^*$-Algorithmus in abgewandelter Form eingesetzt, um unter Zuhilfenahme des Vasilevskii-Chow-Algorithmus ein entsprechendes Modell für ein System zu erlernen.

Literatur

[D.Angluin] D.Angluin(1986), Learning Regular Sets from Queries and Counterexamples

[Pn/Ma] O.Maler,A.Pnueli(2006), On the Learnability of Infinitary Regular Sets

[Gr/Pe/Ya] A.Groce,D.Peled,M.Yannakakis(2006), Adaptive Model Checking

[T.Chow] Tsun S. Chow(1978), Testing Software Design Modeled by Finite-State Machines

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet zu haben.

_____
(Andreas Voetter)