Bachelorarbeit
Universität Bremen

Lösen von Entscheidungsproblemen endlicher Automaten mittels QBF-Solvern

Eingereicht von: Jonas Brenig
jbrenig@uni-bremen.de

Erstgutachter:	Prof. Dr. Thomas Schneider
Zweitgutachter:	Dr. Thomas Barkowsky

04.10.2018

Universität Bremen

Nachname Brenig Matrikelnr. 4237499

Vorname/n Jonas

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum Unterschrift

Erklärung zur Veröffentlichung von Abschlussarbeiten

Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Endliche Automaten spielen in vielen Gebieten der Informatik eine große Rolle. Anwendungen umfassen z. B. Compilerbau, Modellierung oder Model-Checking.

Zwei der bekanntesten Typen von Automaten sind deterministische endliche Automaten (DEA) und nicht-deterministische endliche Automaten (NEA). Obwohl NEAs nicht-deterministisch sind und dadurch mehr Möglichkeiten haben, erkennen beide Typen von Automaten die selbe Klasse von Sprachen [RS59].

Für viele Anwendungen, wie z. B. dem Compilerbau oder dem Parsen von regulären Ausdrücken, werden dabei NEAs verwendet, da diese oft einfacher zu konstruieren sind. Insbesondere sind NEAs oft wesentlich kleiner als äquivalente DEAs. Für die eigentliche Anwendung werden jedoch normalerweise DEAs benötigt, da Software auf existierenden Computern nur deterministisch ausgeführt wird. Aus diesem Grund wird ein NEA oft in einen äquivalenten DEA umgewandelt. Allerdings werden für DEAs im schlimmsten Fall exponentiell größere Automaten benötigt, um die gleiche Sprache zu erkennen [MF71].

Es gibt eine Reihe wichtiger Entscheidungsprobleme für endliche Automaten, die in diesen Anwendungen benötigt werden. Das Wortproblem für endliche Automaten muss beispielsweise entschieden werden, um zu überprüfen ob ein regulärer Ausdruck eine Zeichenkette akzeptiert. Auch andere Probleme wie z. B. Leerheit (ob ein Automat kein Wort akzeptiert), Universalität (ob ein Automat alle Wörter akzeptiert) oder Äquivalenz (ob zwei Automaten die gleichen Wörter akzeptieren) sind hier von Interesse. Für das Leerheitsproblem muss lediglich die Erreichbarkeit der Endzustände geprüft werden. Dieses Problem ist demnach sowohl für DEAs als auch für NEAs effizient lösbar. Für viele andere wichtige Probleme, wie unter anderem Inklusion, Äquivalenz und Universalität, gibt es zwar für DEAs effiziente Verfahren [HK71], jedoch sind diese Probleme für nicht-deterministische endliche Automaten `PSpace`-vollständig [SM73]. Es ist also kein effizienter Algorithmus für Inklusion, Äquivalenz oder Universalität bekannt. Zudem lassen sich diese Probleme aufeinander reduzieren. Ein Entscheidungsverfahren für Inklusion kann mit wenigen Anpassungen verwendet werden, um Universalität oder Äquivalenz zu lösen.

Um Universalität oder Inklusion für nicht-deterministische endliche Automaten zu lösen, wird für gewöhnlich der zugehörige deterministische endliche Automat konstruiert. Dieser wird benötigt, um einen Automaten für das Komplement der Sprache zu bilden, damit dieses auf Leerheit geprüft werden kann. Da dies normalerweise sehr aufwendig ist, gibt es viele Versuche die vollständige Konstruktion dieses Automaten zu vermeiden [DDHR06, BP13, MC13].

In dieser Arbeit werden einige der existierenden Verfahren diskutiert, sowie ein alternativer Ansatz basierend auf quantifizierte boolesche Formel (QBF) vorgestellt und implementiert. Das Erfüllbarkeitsproblem für QBF ist eines der grundlegenden `PSpace`-vollständigen Probleme. Ähnlich zum Erfüllbarkeitsproblem von aussagenlogischen Formeln (SAT)[1], gibt es Verfahren die das Lösen von Formeln für viele Instanzen stark vereinfachen können [Ten16, LE17, KSGC10]. Viele solcher QBF-Solver treten jährlich im Wettkampf[2] gegeneinander an und sind in der Lage sehr große Formeln zu lösen.

Diese Arbeit behandelt eine Kodierung der Probleme Universalität und Inklusion für nicht-deterministische endliche Automaten nach QBF mit dem Ziel die Optimierungen

[1] SAT competition: `http://www.satcompetition.org/`
[2] QBFEval: `http://www.qbflib.org/index_eval.php`

moderner Solver zu verwenden, um diese Probleme in akzeptabler Zeit zu entscheiden. Es soll untersucht werden, wie gut QBF-Solver eine Kodierung der Entscheidungsprobleme Universalität und Inklusion lösen können und wie gut sich dieser Ansatz zu einer direkten Implementierung verhält.

In Kapitel 3 werden zunächst einige bekannte Verfahren für diese Probleme vorgestellt. Als nächstes werden Kapitel 4 mehrere Varianten einer Kodierung nach QBF erläutert und anschließend implementiert (Siehe Kapitel 5). In Kapitel 6 wird schlussendlich ausgewertet, wie gut aktuelle QBF-Solver diese Formeln lösen können und inwiefern diese Ergebnisse mit Implementierungen bekannter Verfahren vergleichbar sind. Teil der Auswertung ist außerdem eine Betrachtung der Größe und Komplexität der erzeugten Formeln in Kapitel 6.2.

2 Grundlagen

Zunächst folgen einige grundlegende Definitionen für endliche Automaten und quantifizierte boolesche Formeln.

2.1 Endliche Automaten

Ein nicht-deterministischer endlicher Automat (NEA) ist ein Tupel $A = (Q, \Sigma, q_1, \Delta, F)$, wobei gilt:

- Q ist eine endliche Menge von Zuständen. Die Zustände des Automaten seien numeriert von 1 bis $|Q|$
- Σ ist ein endliches Alphabet
- $q_1 \in Q$ ist der Startzustand des Automaten
- $\Delta \subseteq Q \times \Sigma \times Q$ sind die Übergänge des Automaten
- $F \subseteq Q$ ist die Menge der Endzustände des Automaten

Falls $(q, a, p) \in \Delta$, dann schreiben wir $q \xrightarrow{a}_A p$. Für die Folge $q_1 \xrightarrow{a_1}_A q_2 \xrightarrow{a_2}_A q_3 \dots q_{n-1} \xrightarrow{a_{n-1}}_A q_n$ mit $w = a_0 \cdot a_1 \cdots a_n$ schreiben wir $q_1 \overset{w}{\Longrightarrow}_A q_n$.

Ein NEA akzeptiert ein Wort w genau dann, wenn $q_1 \overset{w}{\Longrightarrow}_A p$ und $p \in F$. Die erkannte Sprache $\mathcal{L}(A)$ des NEA A besteht aus allen Wörtern $w \in \Sigma^*$, die von A akzeptiert werden. Das zugehörige Komplement $\overline{\mathcal{L}(A)}$ der Sprache ist definiert als $\{w \in \Sigma^* \mid w \notin \mathcal{L}(A)\}$

Für die Länge eines Wortes w schreiben wir $|w|$. Das Wort der Länge $|w| = 0$ wird notiert als ε.

Für endliche Automaten kann eine grafische Darstellung verwendet werden. Dabei wird der Automat als Graph repräsentiert. Die Knoten des Graphen repräsentieren die Zustände des Automaten und sind gegebenenfalls durch eine doppelte Umrandung markiert, falls es sich um Endzustände handelt. Die Übergänge des Automaten werden durch Kanten im Graphen dargestellt. Der Startzustand des Automaten wird durch eine eingehende, *in der Luft hängende* Kante markiert. Ein einfacher Automat mit 4 Zuständen und einigen Kanten könnte so aussehen (Abbildung 2.1):

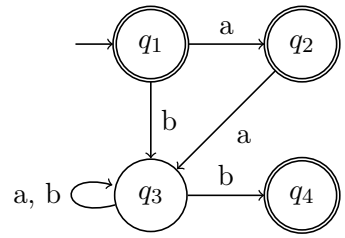


Abbildung 2.1: Beispielautomat mit 4 Zuständen

2.2 Deterministische endliche Automaten

Ein NEA heißt deterministischer endlicher Automat (DEA) gdw. :

$$\forall q \in Q \; \forall a \in \Sigma : \; |\{(q, a, p) \in \Delta \mid p \in Q\}| = 1$$

Zu jedem NEA A gibt es einen DEA A^d, mit $\mathcal{L}(A) = \mathcal{L}(A^d)$ [RS59].
Dieser Automat A^d kann mithilfe der Potenzmengenkonstruktion erstellt werden.
Dann sei $A^d = (Q', \Sigma, \{q_1\}, \Delta', F')$, wobei

- $Q' = 2^Q$
- $\Delta' = \{(p, a, p') \mid p' = \{q' \in Q \mid \exists q \in p : \; (q, a, q') \in \Delta\}\}$
- $F' = \{p \in Q' \mid \exists q \in p : \; q \in F\}$

Jeder Zustand p in A^d hat genau einen Übergang $(p, a, p') \in \Delta'$ für jeden Buchstaben $a \in \Sigma$.

2.3 Entscheidungsprobleme endlicher Automaten

Es gibt viele Entscheidungsprobleme für endliche Automaten. In einer Arbeit von Stockmeyer und Meyer aus dem Jahr 1973 [SM73] wird die Komplexität einiger dieser Probleme diskutiert. Einen Überblick über Entscheidungsprobleme endlicher Automaten gibt außerdem eine Arbeit von Holzer und Kutrib aus dem Jahr 2009 [HK09]. Im Folgenden werden einige der Entscheidungsprobleme für NEAs und DEAs vorgestellt. Während viele Probleme für DEAs effizient, also in polynomieller Zeit oder besser entschieden werden können, sind einige der Probleme für NEAs um einiges schwerer.

Wortproblem

Gegeben sei ein Automat A und ein Wort w, entscheide ob $w \in \mathcal{L}(A)$.

Das Wortproblem ist sowohl für DEAs, als auch für NEAs effizient lösbar.

Leerheit

Gegeben sei ein Automat A, entscheide ob $\mathcal{L}(A) = \emptyset$.

Dieses Problem ist sowohl für NEAs und DEAs effizient zu lösen, da ein minimal langes Wort $w \in \mathcal{L}(A)$ maximal aus $|Q|$ Buchstaben besteht.

Universalität

Gegeben sei ein Automat A, entscheide ob $\mathcal{L}(A) = \Sigma^*$. Wenn Universalität nicht gilt, gibt es ein Wort w mit $w \notin \Sigma^*$. Ein solches Wort w wird auch Gegenbeispiel für Universalität genannt.

Während dieses Problem für DEAs effizient zu lösen ist [HK71], ist Universalität für NEAs in `PSpace` [SM73].

Inklusion und Äquivalenz

Gegeben seien zwei Automaten A und B, entscheide ob $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. Inklusion kann dabei auf Äquivalenz ($\mathcal{L}(A) = \mathcal{L}(B)$) erweitert werden, indem auf beidseitige Inklusion geprüft wird ($\mathcal{L}(A) \subseteq \mathcal{L}(B) \wedge \mathcal{L}(B) \subseteq \mathcal{L}(A)$).

Wenn Inklusion nicht gilt, gibt es ein Wort w mit $w \in \mathcal{L}(A) \wedge w \notin \mathcal{L}(B)$. Ein solches Wort w wird auch Gegenbeispiel für Inklusion genannt.

Ähnlich zu Universalität ist dieses Problem für NEAs schwierig, während für DEAs ein effizienter Algorithmus bekannt ist [HK71].

Minimierung

Minimierung von Automaten ist ein Berechnungsproblem. Gegeben sei ein Automat A, gesucht ist ein äquivalenter Automat A' mit minimaler Zustandszahl.

Minimierung von NEAs ist ein schwieriges Problem. Das zugehörige Entscheidungsproblem ist ebenfalls `PSpace`-vollständig [JR93].

Für die Minimierung von DEAs hingegen ist ein effizienter Algorithmus bekannt [Hop71].

2.4 Quantifizierte boolesche Formeln

Eine QBF ist eine aussagenlogische Formel, in der die Quantoren $\exists, \forall$ es ermöglichen über Variablen zu quantifizieren.

Eine QBF ist dabei wie folgt aufgebaut:

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \ \varphi$$

Wobei $Q_i \in \{\exists, \forall\}$ und φ eine aussagenlogische Formel ist.

Um die Konstruktion von QBFs zu erleichtern, kann diese Definition angepasst werden, sodass die Schachtelung von Quantoren erlaubt ist. Der Aufbau dieser generalisierten QBF ist dabei wie folgt induktiv definiert:

- Die Konstanten `true` und `false` sind QBF
- Eine Variable x ist eine QBF
- Sei φ eine QBF, dann auch $\exists x \ \varphi$
- Sei φ eine QBF, dann auch $\forall x \ \varphi$
- Sei φ eine QBF, dann auch $\neg\varphi$
- Seien φ_1 und φ_2 QBF, dann auch $\varphi_1 \wedge \varphi_2$
- Seien φ_1 und φ_2 QBF, dann auch $\varphi_1 \vee \varphi_2$

Weitere logische Operatoren, wie z. B. $\rightarrow$ können mit den bereits eingeführten Operatoren ausgedrückt werden.

Eine *Belegung* β weist jeder Variable x entweder den Wert $\beta(x) = \texttt{true}$ oder den Wert $\beta(x) = \texttt{false}$ zu. Wir schreiben $\beta[x/\texttt{true}]$ für die Belegung, die alle Variablen y auf den alten Wert $\beta[x/\texttt{true}](y) := \beta(y)$ setzt, bis auf die Variable x für die gilt $\beta[x/\texttt{true}](x) = \texttt{true}$. (Analog für $\beta[x/\texttt{false}]$). $\beta(\varphi)$ wird für komplexere QBF induktiv über die Struktur der Formel wie folgt erweitert:

$$\beta(\psi_1 \wedge \psi_2) = \begin{cases} \texttt{true} & \text{falls } \beta(\psi_1) = \texttt{true} \text{ und } \beta(\psi_2) = \texttt{true} \\ \texttt{false} & \text{sonst} \end{cases}$$

$$\beta(\psi_1 \vee \psi_2) = \begin{cases} \texttt{true} & \text{falls } \beta(\psi_1) = \texttt{true} \text{ oder } \beta(\psi_2) = \texttt{true} \\ \texttt{false} & \text{sonst} \end{cases}$$

$$\beta(\neg\psi) = \begin{cases} \texttt{true} & \text{falls } \beta(\psi) = \texttt{false} \\ \texttt{false} & \text{sonst} \end{cases}$$

$$\beta(\forall x \ \psi) = \begin{cases} \texttt{true} & \text{falls } \beta[x/\texttt{true}](\psi) = \texttt{true} \text{ und } \beta[x/\texttt{false}](\psi) = \texttt{true} \\ \texttt{false} & \text{sonst} \end{cases}$$

$$\beta(\exists x \ \psi) = \begin{cases} \texttt{true} & \text{falls } \beta[x/\texttt{true}](\psi) = \texttt{true} \text{ oder } \beta[x/\texttt{false}](\psi) = \texttt{true} \\ \texttt{false} & \text{sonst} \end{cases}$$

Ein Variablenvorkommen von x heißt frei in φ, wenn die Variable x in der Formel φ nicht von einem Quantor gebunden wurde. D. h. es gibt ein Vorkommen von x in φ außerhalb von Teilformeln der Form $\exists x\ \psi$ oder der Form $\forall x\ \psi$.

Eine QBF $\varphi(x_1, ..., x_n)$, mit freien Variablen $x_1, ..., x_n$ heißt *erfüllbar* genau dann, wenn es eine Belegung β gibt, sodass $\beta(\varphi) = \texttt{true}$. Wir schreiben auch $\beta \models \varphi$. Die Formel φ heißt *gültig* genau dann, wenn für jede Belegung β gilt: $\beta \models \varphi$. Bei QBFs ohne freie Variablen ist *gültig* und *erfüllbar* also gleichbedeutend.

Eine QBF ist in Pränexnormalform, falls alle verwendeten Quantoren am Beginn der Formel stehen. Also wenn gilt $\varphi = Q_1 x_1 ... Q_n x_n\ \psi$, mit $Q_i \in \{\exists, \forall\}$, x_i eine beliebige Variable und ψ ist eine Formel ohne Quantoren.

Eine Formel ist in konjunktiver Normalform (KNF), falls φ in Pränexnormalform und ψ in der Form $\bigwedge_i \bigvee_j l_{ij}$ mit $l_{ij} = \begin{cases} x_{ij} \\ \neg x_{ij} \end{cases}$, wobei x_{ij} eine beliebige Variable. Analog gilt bei disjunktiver Normalform (DNF): $\psi = \bigvee_i \bigwedge_j l_{ij}$.

3 Bekannte Techniken

Sowohl Inklusion, als auch Universalität von nicht-deterministischen endlichen Automaten sind PSpace-vollständig [SM73]. Im Allgemeinen sind also keine effizienten Algorithmen bekannt, die diese Probleme entscheiden.

Ein sehr einfacher PSpace-Algorithmus um Inklusion zu entscheiden, basiert auf der Potenzmengenkonstruktion. Es gibt jedoch auch einige andere Ansätze, die für viele Instanzen wesentlich schneller sind [DDHR06, BP13].

Eine Sammlung aktueller Forschungsresultate in diesem Bereich ist auf der Website http://languageinclusion.org verfügbar.

In diesem Kapitel werden eine Reihe bekannter Techniken vorgestellt. Dabei wird diskutiert, welche Verfahren sich für eine Kodierung in QBF eignen.

3.1 Universalität mittels Potenzmengenkonstruktion

Der einfachste Ansatz Universalität (und Inklusion) zu entscheiden verwendet die Potenzmengenkonstruktion.

Seien A NEAs. Wenn $\mathcal{L}(A) = \Sigma^*$, dann gilt $\forall w \in \Sigma^* : \ w \in \mathcal{L}(A)$.

Es muss also gezeigt werden, dass $\overline{\mathcal{L}(A)} = \emptyset$

Zu diesem Zweck wird der Automat A mithilfe der Potenzmengenkonstruktion in einen deterministischen Automaten A^d umgewandelt (Siehe Kapitel 2.2).

Für jedes Wort $w \in \Sigma^*$ muss jetzt geprüft werden, ob $w \in \overline{\mathcal{L}(A)}$. Wird ein solches Wort gefunden, ist A nicht universell. Damit dieses Verfahren terminiert ist wichtig, dass nur endliche viele Gegenbeispiele geprüft werden müssen (Siehe Lemma 4.1.1).

Um zu entscheiden, ob $w \in \mathcal{L}(A)$ muss nicht der gesamte, potentiell exponentiell größere, Potenzautomat vorliegen. Es genügt, den aktuellen Zustand des Potenzautomaten zu speichern. Der nächste Zustand für die Simulation von A^d unter w kann bei Bedarf berechnet werden.

Ein einfacher nicht-deterministischer Algorithmus um zu entscheiden, ob ein Automat nicht universell ist, benötigt nur polynomiell viel Platz.

Algorithm 1 coNPSpace Algorithmus für Universalität

procedure NichtUniversell
  $M := \{q_1\}$
  for $0 \leq i < 2^{|Q|}$ **do**
    Rate Zustandsmenge $N \subseteq Q$
    if $M \neq N$ und $\nexists a \in \Sigma : \ M \xrightarrow{a}_{A^d} N$ **then return** `false`
    $M := N$
  return $\forall q \in F : \ q \notin M$

Da PSpace = NPSpace = coNPSpace gibt es für diesen Ansatz einen PSpace-Algorithmus [Sav70]. Dementsprechend ist dieses Verfahren für eine Kodierung in QBF geeignet.

Um einen solchen Algorithmus zu implementieren muss allerdings eine deterministische Variante verwendet werden. Daher bietet sich aufbauend auf der Idee von Savitch der Algorithmus 2 an.

Es ist außerdem möglich, diesen Ansatz auf Inklusion (und damit auch auf Äquivalenz) zu erweitern. Für den Test zur Inklusion $B \subseteq A$ müssen die betrachteten Kandidaten

Algorithm 2 `coNPSpace` Algorithmus für Universalität

procedure NICHTUNIVERSELL
  for $M \subseteq Q \setminus F$ **do**
    if $Reach(\{q_1\}, M, |Q|)$ **then return** `true`
  return `false`
procedure REACH(From, To, i)
  if $i = 0$ **then return** $From = To$ oder $\exists a \in \Sigma : \ From \xrightarrow{a}_{A^d} To$
  for $M \subseteq Q$ **do**
    if $Reach(From, M, i-1) \wedge Reach(M, To, i-1)$ **then return** `true`
  return `false`

für Gegenbeispiele zusätzlich die Bedingung $w \in \mathcal{L}(B)$ erfüllen. Gesucht sind also Wörter $w \notin \mathcal{L}(A) \wedge w \in \mathcal{L}(B)$. Für Inklusion müssen zwar eventuell mehr Gegenbeispiele geprüft werden, jedoch reicht es weiterhin nur endlich viele Wörter zu untersuchen (Siehe Lemma 4.2.1).

Eine existierende Implementierung dieses grundlegenden Ansatzes ist in dem Programm GOAL[1] [TCT+08] enthalten. GOAL erzeugt zu diesem Zweck als erstes das Komplement des Eingabeautomaten und testet anschließend auf Leerheit. GOAL ist in erster Linie entworfen um mit temporalen Logiken und Büchi-Automaten (eine Art endlicher Automaten die Wörter unendlicher Länge verwendet) zu arbeiten, kann jedoch unter anderem auch für NEAs verwendet werden.

3.2 Universalität mittels Antiketten

Wulf, Doyen, Henzinger und Raskin haben im Jahr 2006 einen Algorithmus auf Basis von Antiketten vorgestellt, der Inklusion von NEAs für viele Eingaben schneller entscheiden kann als der im vorherigen Kapitel vorgestellte klassische Potenzmengen-Algorithmus [DDHR06].

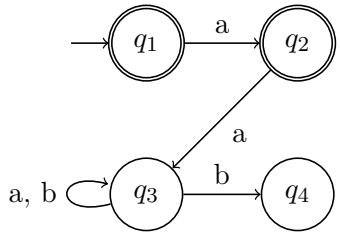
Eine Menge F ist dabei genau dann eine Antikette, wenn es keine zwei Mengen $S \in F$ und $S' \in F$ gibt, sodass $S \subseteq S'$.

Ausgehend von allen nicht akzeptierenden Zuständen, wird es versucht den Startzustand zu erreichen. Falls der Startzustand jemals erreicht wird, ist der Automat nicht universell.

In einer Menge von Zustandsmengen F wird gespeichert, welche Zustände auf diese Weise in A erreichbar sind. Gleichzeitig wird in der Menge von Zustandsmengen $Frontier$ gespeichert, welche Zustandsmengen zuletzt gefunden wurden, also welche Zustandsmengen im nächsten Iterationsschritt betrachtet werden müssen.

In jeder Iteration des Algorithmus wird $Frontier$ neu berechnet und mit F (unter Erhaltung der Antikette) vereinigt. Solange es eine neue Zustandsmenge M und eine Zustandsmenge $S \in Frontier$ gibt, sodass jeder der enthaltenen Zustände $q \in M$ unter einem Buchstaben a nur Übergänge in Zustände $q' \in S$ hat, wird diese Menge der neuen $Frontier$ Menge hinzugefügt. D. h. für jede Menge $M \in Frontier$ gilt, dass der zugehörige Potenzautomat A^d von einem Zustand $M' \subseteq M$ keinen Endzustand erreichen kann.

[1] GOAL: `http://goal.im.ntu.edu.tw/wiki/doku.php`



1. $F = Frontier = \{\{q_3, q_4\}\}$
2. $Frontier = \{\{q_2, q_3\}\}$
   $F = \{\{q_2, q_3\}, \{q_3, q_4\}\}$
3. $Frontier = \{\{q_1, q_2, q_3\}\}$
   $F = \{\{q_1, q_2, q_3\}, \{q_3, q_4\}\}$
4. $\{\{q_1, q_2, q_3\}\} \in Frontier$ enthält den Startzustand, also terminiert der Algorithmus und gibt `false` aus.

Abbildung 3.1: Beispiel für den Antiketten-Algorithmus

Außerdem wird sichergestellt, dass sowohl F als auch $Frontier$ die Antiketten Bedingung erfüllen. Dadurch müssen viele Zustandsmengen von dem Antiketten-Algorithmus nicht weiter betrachtet werden. Dies verhindert nicht, dass alle möglichen Pfade gefunden werden.

Dieses Verfahren lässt sich erweitern auf Inklusion [DDHR06]. Da die Antiketten im schlimmsten Fall exponentiell viel Platz benötigen, ist dieser Algorithmus allerdings kein `PSpace` Algorithmus. Für viele Eingaben ist dieses Verfahren dennoch sehr effizient [DDHR06].

Der Antiketten Algorithmus wurde in der Bibliothek VATA[2] implementiert.

3.3 Äquivalenz mittels Bisimulation

Im Jahr 2013 stellten Bonchi und Pous einen Algorithmus zum Entscheiden der Äquivalenz von NEAs vor [BP13]. In diesem Ansatz werden Bisimulationen verwendet, um Äquivalenz von Automaten zu zeigen. Eine Bisimulation für einen DEA A ist eine Relation $R \subseteq Q \times Q$, sodass für alle $q, q' \in Q$ gilt:

- $\forall a \in \Sigma \ : \ (q, q') \in R$ und die Nachfolger von q und q' unter dem Buchstaben a sind enthalten in R.
- $q \in F \ gdw. \ q' \in F$

Zwei Automaten sind äquivalent, wenn es eine Bisimulation gibt, sodass die Anfangszustände der beiden Automaten in Relation zu einander stehen. Die Definition von Bisimulation umfasst nur einen Automaten, wenn zwei Automaten A, B verwendet werden müssen diese also als Einheit aufgefasst werden. Für die Bisimulation kann dafür die disjunktive Vereinigung der benötigten Mengen $Q = Q_A \uplus Q_B$, $\Delta = \Delta_A \uplus \Delta_B$, ... verwendet werden.

Dieser Ansatz lässt sich außerdem leicht auf NEAs erweitern, in dem die Konstruktion des Potenzautomaten direkt integriert wird. Eine Bisimulation für einen NEA A ist eine Relation $R \subseteq P(Q) \times P(Q)$, sodass für alle $q, q' \in P(Q)$ gilt:

- $\forall a \in \Sigma \ : \ (q, q') \in R$ und die Nachfolger von q und q' unter dem Buchstaben a im zugehörigen Potenzautomaten A^d sind enthalten in R.
- $q \cap F = \emptyset \ gdw. \ q' \cap F = \emptyset$

Bonchi und Pous erweitern diesen Ansatz um Funktionen f, die auf diese Relation angewandt werden und weiterhin für eine Bisimulation verwendet werden können. In dem von

[2] libVATA: `https://github.com/ondrik/libvata`

ihnen vorgestellten Algorithmus wird versucht, ausgehend von den beiden Startzuständen der Automaten, eine Bisimulation R aufzubauen. Für jedes betrachtetes Paar von Zuständen (x, y) wird geprüft, ob dieses bereits in $f(R)$ enthalten ist. Je nach gewählter Funktion f muss so nicht der gesamte Automat durchsucht werden.

In dem Paper wird eine solche Funktion für NEAs vorgestellt, für die große Teile des Potenzautomaten nicht besucht werden müssen. Dadurch kann Äquivalenz für viele Eingaben schneller gelöst werden, als mit der klassischen Potenzmengenkonstruktion. In der angesprochenen Arbeit wird Bisimulation außerdem mit dem Antiketten-Algorithmus verglichen. Weitergehend wird das Antiketten-Verfahren sogar als Variante der Bisimulation erläutert. Im dort anschließend durchgeführten Vergleich ist der vorgestellte Algorithmus (im Paper beschrieben als *Bisimulation up-to congruence*) für die meisten Fälle effizienter als der Antiketten-Algorithmus.

Auch eine Bisimulation benötigt im schlimmsten Fall exponentiell viel Platz und liefert dementsprechend keinen `PSpace` Algorithmus. Die im Paper von Bonchi und Pous verwendete Implementation dieses Algorithmus kann online[3] abgerufen werden.

3.4 Minimierung von Automaten

Die Größe der Eingabeautomaten ist ausschlaggebend für die Performanz von Algorithmen die Inklusion entscheiden sollen. Effiziente Algorithmen, welche die Größe der Automaten reduzieren, sind also eine interessante Möglichkeit die bereits vorgestellten Algorithmen zu beschleunigen. Die Minimierung von NEAs ist im Allgemeinen allerdings sehr aufwendig.

Mayr und Clemente [MC13] stellen zu diesem Zweck einen Ansatz vor, um Eingabeautomaten heuristisch zu verkleinern. In ihrem Paper wird zwar mit Büchi-Automaten gearbeitet, das Verfahren lässt sich jedoch auch für NEAs verwenden.

Büchi-Automaten unterscheiden sich von NEAs lediglich durch ihre Akzeptanzbedingung, da sie auf unendlich langen Wörtern arbeiten. Ein unendliches Wort w wird dabei von einem Büchi-Automaten genau dann akzeptiert, wenn während der Berechnung unendlich oft Endzustände erreicht werden.

Die meisten der vorgestellten Verarbeitungsschritte sind in der Lage viele Automaten zu vereinfachen, ohne die erkannte Sprache zu verändern. Einige der Ansätze schwächen diese Garantie etwas ab und stellen nur sicher, dass eine Inklusion erhalten bleiben würde.

In dem Tool RABIT[4] [MC13] ist dieses Verfahren zum Lösen von Inklusion implementiert. (Sowohl für Büchi-Automaten als auch für NEAs)

Durch die Verkleinerung der Eingabeautomaten kann Inklusion bereits in vielen Fällen direkt entschieden werden, da triviale Automaten mit nur einem Zustand erzeugt werden. Ansonsten kann ein anderes Verfahren, wie z. B. Antiketten oder Bisimulation, auf der minimierten Eingabe ausgeführt werden.

Dabei wird in vier Schritten vorgegangen: Nach der Minimierung des Automaten wird ein im Paper eingeführtes Simulationsverfahren angewendet, das ebenfalls in polynomieller Zeit läuft. Anschließend wird mit entsprechendem Timeout versucht, ein kleines Gegenbeispiel zu finden. Falls keiner dieser drei Schritte Inklusion erfolgreich entscheiden kann, wird ein vollständiges Verfahren (wie z. B. Antiketten) verwendet.

Durch diese Vorgehensweise ist das Tool RABIT in der Lage wesentlich größere Instanzen zu lösen als andere Verfahren. Da vollständige Verfahren stark von einer Minimierung der Eingabeautomaten profitieren, ist es sinnvoll vollständige Verfahren nur als Schritt 4 zu verwenden.

[3] Bisimulation; `http://perso.ens-lyon.fr/damien.pous/hknt/`
[4] RABIT: `www.languageinclusion.org/doku.php?id=tools`

4 Kodierung als QBF

Um Universalität und Inklusion mithilfe von QBF-Solvern zu entscheiden, müssen diese Probleme in QBF kodiert werden. Als Grundlage dient hierbei das Entscheidungsverfahren mittels Potenzmengenkonstruktion (Siehe Kapitel 2.2). Bei der Kodierung als QBF muss beachtet werden, dass die Konstruktion nicht zu viel Platz verwendet.

4.1 Universalität

Um Universalität mit QBF zu entscheiden, wird die Idee der Potenzmengenkonstruktion aus Kapitel 3.1 in die Formel kodiert. Gesucht ist ein Pfad im Potenzautomaten A^d, vom Anfangszustand q_1 zu einem Zustand $q \notin F$. Existiert ein solcher Pfad, gibt es ein Wort w mit $q_1 \overset{w}{\Longrightarrow}_{A^d} q$, das nicht in der Sprache des Automaten enthalten ist. Der Automat ist genau dann nicht universell.

Damit dieses Verfahren terminiert, dürfen nur endlich viele Wörter w untersucht werden. Hier hilft folgende Beobachtung.

Lemma 4.1.1. *Gegeben ein DEA A. $\forall w \notin \mathcal{L}(A)$ gilt: $\exists w' \notin \mathcal{L}(A)$, sodass: $|w'| \leq |Q|$*

Beweis. Ähnlich zum klassischen Pumping Lemma [BHPS61] lässt sich auch hier argumentieren:

Angenommen $\mathcal{L}(A) \neq \Sigma^*$. Sei $w \notin \mathcal{L}(A)$ und $|w| > |Q|$ minimal. Dann gibt es einen Zustand $q \in Q$, sodass $q_1 \overset{u}{\Longrightarrow}_A q \overset{k}{\Longrightarrow}_A q \overset{v}{\Longrightarrow}_A q'$, wobei $w = ukv, k \neq \varepsilon$ und $q' \notin F$. Da A deterministisch, gilt $w' = uv \notin \mathcal{L}(A)$. Offensichtlich ist $|w|$ nicht minimal. □

Da der Potenzautomat potentiell exponentiell groß wird, kann er nicht direkt als Formel kodiert werden. Daher wird der Potenzautomat *on the fly* konstruiert. D. h. nur die für Gegenbeispiele (Wörter mit $w \notin \mathcal{L}(A)$) benötigten Zustandsmengen werden in der Formel repräsentiert (Siehe Lemma 4.1.5). Diese Zustandsmengen M werden als logische Variablen $x_1, ..., x_n$ kodiert. In der Belegung soll dabei gelten $\beta(x_i) = 1$ *gdw.* $q_i \in M$.

4.1.1 Notation

Sei $|Q| = n$. Für die Kodierung einer Zustandsmenge M des Potenzautomaten A^d wird ein n-stelliges Tupel $X = (x_1, ..., x_n)$ verwendet.

Eine Quantifizierung von $\exists X$ stellt also die Quantifizierung von n aussagenlogischen Variablen dar: $\exists x_1, ..., \exists x_n$. Wir schreiben $X \leftrightarrow Y$ um folgende Formel auszudrücken: $x_1 \leftrightarrow y_1 \wedge ... \wedge x_n \leftrightarrow y_n$.

Im Folgenden wird gegebenenfalls zwischen der Repräsentation als Menge von Zuständen und der als logisches Tupel gewechselt. Gegeben das logische Tupel X, dann sei $\texttt{state}_\beta(X)$ definiert als: $\texttt{state}_\beta(X) = M$, mit $M \subseteq Q$ und $q_i \in M$ *gdw.* $\beta(x_i) = \texttt{true}$.

4.1.2 Konstruktion

Um Universalität als QBF zu kodieren, werden zunächst einige Formeln benötigt, um bestimmte Eigenschaften des Automaten auszudrücken.

Zum Festlegen des Startzustandes q_1 des Automaten A wird die Formel $initial_A$ definiert.

$$initial^A(X) = x_1 \wedge \bigwedge_{\substack{q_i \in Q \\ q_i \neq q_1}} (\neg x_i) \tag{4.1}$$

Außerdem muss kodiert werden, dass es einen Übergang zwischen zwei Zustandsmengen im Potenzautomaten A^d gibt. Die Formel $trans_a^A(X,Y)$ bedeutet also: $\texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$

$$trans_a^A(X,Y) = \bigwedge_{i \leq n} (y_i \leftrightarrow \bigvee_{\substack{j \leq n \\ (q_j,a,q_i) \in \Delta_A}} x_j) \tag{4.2}$$

Als nächstes erweitert auf einen beliebigen Buchstaben, bedeutet $trans_\Sigma^A(X,Y)$, dass $\texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$ für ein $a \in \Sigma$.

$$trans_\Sigma^A(X,Y) = \bigvee_{a \in \Sigma} trans_a^A(X,Y) \tag{4.3}$$

Um festzustellen, ob die Zustandsmenge Y von X in einem oder weniger Schritten erreichbar ist, wird die Formel $reach_0^A(X,Y)$ benötigt. Diese Formel bedeutet also, dass $\texttt{state}_\beta(X) = \texttt{state}_\beta(Y)$ oder $\texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$ für ein $a \in \Sigma$.

$$reach_0^A(X,Y) = (X \leftrightarrow Y) \vee trans_\Sigma^A(X,Y) \tag{4.4}$$

Gegeben eine Belegung β soll die Formel $reach_n^A(X,Y)$ von β erfüllt werden gdw. die zugehörige Zustandsmenge $\texttt{state}_\beta(Y)$ von $\texttt{state}_\beta(X)$ im Automaten A^d in $\leq 2^n$ Schritten erreichbar ist. D. h. es gibt ein Wort w mit $X \overset{w}{\Longrightarrow}_{A^d} Y$ im Potenzautomaten A^d.

Jeder deterministische endliche Automat (DEA) mit $\mathcal{L}(A) \neq \Sigma^*$ hat Gegenbeispiele w mit $|w| \leq |Q|$ (Siehe Lemma 4.1.1). Da der deterministische Potenzautomat A^d eines NEAs A maximal $2^{|Q|}$ Zustände hat, müssen für NEAs nur Gegenbeispiele der Länge $2^{|Q|}$ betrachtet werden.

Um zu vermeiden, dass der gesamte Potenzautomat in der Formel konstruiert werden muss, wird eine Zustandsmenge H geraten, welche den Pfad im Potenzautomaten A^d *mittig* teilt. Hier hilft die Beweisidee des Satzes von Savitch [Sav70]. Also gilt $X \overset{u}{\Longrightarrow}_{A^d} H \overset{v}{\Longrightarrow}_{A^d} Y$, mit $uv = w, |u| \leq 2^{n-1}, |v| \leq 2^{n-1}$.

Damit die Formel durch die exponentiell vielen *Aufrufe* von $reach_i^A$ nicht exponentiell groß wird, werden die Tupel logischer Variablen L und R mittels $\forall$-Quantor quantifiziert. Der linke Teil der Formel kann von einer Belegung β nur unter zwei Bedingungen erfüllt werden. Die Formel $reach_{n-1}^A(L,R)$ muss dann von β sowohl für $(\beta(L) = \beta(X))$ und $(\beta(R) = \beta(H))$, als auch für $(\beta(L) = \beta(H))$ und $(\beta(R) = \beta(Y))$ erfüllt werden.

$$reach_i^A(X,Y) = \exists M \forall L \forall R \big((X \leftrightarrow L \wedge M \leftrightarrow R) \vee (M \leftrightarrow L \wedge Y \leftrightarrow R) \big) \rightarrow reach_{i-1}^A(L,R) \tag{4.5}$$

Nun kann eine QBF konstruiert werden, die erfüllbar ist gdw. $L(A) \neq \Sigma^*$:

$$notuniv = \exists X \exists Y \; initial^A(X) \wedge (\bigwedge_{q_i \in F} \neg y_i) \wedge reach_n^A(X,Y) \tag{4.6}$$

Für Universalität wird nur die Negation dieser Formel benötigt:

$$univ = \neg notuniv \tag{4.7}$$

4.1.3 Korrektheit

Zunächst ist es zu zeigen, dass die vorgestellte Kodierung korrekt ist. Also ob die für einen NEA A konstruierte Formel *notuniv* genau dann erfüllbar ist, wenn $\exists w : w \notin \mathcal{L}(A)$. Anschließend wird der Platzverbrauch der generierten Formel betrachtet.

Zu diesem Zweck werden beginnend mit der Teilformel $trans_a^A$ zuerst einige Eigenschaften der Teilformeln gezeigt.

Lemma 4.1.2. *Für alle NEAs A und Belegungen β gilt:*
$\beta \models trans_a^A(X,Y)$ *gdw.* $\texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$

Beweis. Sei $\beta \models trans_a^A(X,Y)$. Genau dann gilt für alle y_i mit $i \leq n$, dass $\beta(y_i) = \texttt{true}$ *gdw.* $\exists x_j : j \leq n$ und $(q_j, q_i, a) \in \Delta$ und $\beta(x_j) = \texttt{true}$. Da jede logische Variable direkt einem Zustand $q_i \in Q$ entspricht, folgt direkt $y \in \texttt{state}_\beta(Y)$ *gdw.* $\exists x \in \texttt{state}_\beta(X) : (x, a, y) \in \Delta$. Nach Definition des Potenzautomaten (Siehe Kapitel 2.2) gilt genau dann also $\texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$. □

Erweitert auf eine Transition mit einem beliebigen Buchstaben $a \in \Sigma$ kann jetzt gezeigt werden:

Lemma 4.1.3. *Für alle NEAs A und Belegungen β gilt:*
$\beta \models trans_\Sigma^A(X,Y)$ *gdw.* $\exists a \in \Sigma : \texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$

Beweis. $\beta \models trans_\Sigma^A(X,Y)$ genau dann, wenn es ein $a \in \Sigma$ gibt, sodass $\beta \models trans_a^A(X,Y)$. Nach Lemma 4.1.2 gilt also: $\beta \models trans_\Sigma^A(X,Y)$ genau dann, wenn es ein $a \in \Sigma$ gibt, sodass $\texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$. □

Lemma 4.1.4. *Für alle NEAs A und Belegungen β gilt:*
$\beta \models reach_0^A(X,Y)$ *gdw.* $\exists a \in \Sigma : \texttt{state}_\beta(X) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y)$ *oder* $\texttt{state}_\beta(X) = \texttt{state}_\beta(Y)$

Beweis. Offensichtlich ist die Formel genau dann erfüllt, wenn entweder

$$\texttt{state}_\beta(X) = \texttt{state}_\beta(Y)$$

oder es einen Übergang im Automaten zwischen diesen beiden Zuständen gibt. □

Als nächstes kann gezeigt werden, dass die Teilformel $reach_i^A$ ein Wort der Länge $|w| \leq 2^i$ beschreibt:

Lemma 4.1.5. *Für alle NEAs A und Belegungen β gilt:*
$\beta \models reach_i^A(X,Y)$ *gdw.* $\exists w \in \Sigma^* : |w| \leq 2^i$ *und* $\texttt{state}_\beta(X) \xRightarrow{w}_{A^d} \texttt{state}_\beta(Y)$

Beweis. Beweis per Induktion über i. Der Induktionsanfang für $i = 0$ gilt nach Lemma 4.1.4. Die Aussage sei bereits gezeigt für $i-1$. Für den Induktionsschritt $i-1 \to i$ gilt also:

Es gibt nur zwei Möglichkeiten die allquantifizierten Tupel von Variablen L und R zu belegen, sodass der linke Teil der Implikation gilt. (Siehe Formel 4.5)

Entweder gilt $\beta(X) = \beta(L)$ und $\beta(M) = \beta(R)$ oder $\beta(M) = \beta(L)$ und $\beta(Y) = \beta(R)$. Es gilt also $\beta \models reach_i^A(X,Y)$ genau dann, wenn $\beta' \models reach_{i-1}^A(L,R)$, für diese beiden Möglichkeiten L und R zu belegen. (Hierbei sei β' eine von β angepasste Belegung, die L und R entsprechend belegt.)

Nach Induktionsvoraussetzung folgt also, dass $\beta \models reach_i(X,Y)$ genau dann, wenn $\texttt{state}_\beta(X) \xRightarrow{u}_{A^d} \texttt{state}_\beta(M) \xRightarrow{v}_{A^d} \texttt{state}_\beta(Y)$ und $|u| \leq 2^{i-1}$ und $|v| \leq 2^{i-1}$ und $uv = w$. Offensichtlich kann M so gewählt werden, dass auch ein bis zu 2^i langes Wort gefunden wird. (Dann muss $|u| = 2^{i-1}$ und $|v| = 2^{i-1}$) □

Um das Hauptresultat zu zeigen, wird zusätzlich zu Lemma 4.1.2 - 4.1.5 folgende Beobachtung benötigt.

Beobachtung 4.1.6. *$initial_A(X)$ hat offensichtlich nur eine erfüllende Belegung. Diese Belegung repräsentiert die Zustandsmenge $M = \{q_1\} = \texttt{state}_\beta(X)$, wobei q_1 der Anfangszustand des Automaten A ist.*

Unter Verwendung dieser Beobachtung und Lemma 4.1.5 kann nun gezeigt werden, dass:

Theorem 4.1.7. *Für alle NEAs A gilt:*
notuniv ist gültig gdw. $\exists w \in \Sigma^* : w \notin L(A)$

Beweis. Der erste Teil der Formel stellt sicher, dass $\texttt{state}_\beta(X) = \{q_1\}$ der Startzustand des Automaten ist. Aus dem zweiten Teil ergibt sich, dass $\texttt{state}_\beta(Y)$ kein Endzustand von A ist. Die Belegung von X und Y kann offensichtlich fast immer so gewählt werden, dass diese beiden Teilformeln erfüllt sind. Die einzige Ausnahme ist ein NEA A, bei dem A^d keine nicht-Endzustände hat. Ein solcher NEA ist allerdings offensichtlich universell.

Nach Lemma 4.1.5 erfüllt eine Belegung β die Teilformel $reach^{B\subseteq A}_{n\cdot\lceil log(m)\rceil}$ genau dann, wenn es einen entsprechenden Pfad in beiden Automaten gibt. (d. h. $\exists\beta : \ \beta \models reach^A_n$) Also gibt es genau dann ein Gegenbeispiel w mit $|w| \leq n$, wenn die Formel *notuniv* erfüllbar ist.

Nach Lemma 4.1.1 gibt es auch nur genau dann ein Gegenbeispiel w von beliebiger Länge. □

Platzverbrauch der Formel

Zuletzt muss die Komplexität der Konstruktion analysiert werden.

Die erzeugte Formel besteht aus drei Komponenten. Die Teilformeln $initial^A$ und $\bigwedge_{q_i \in F} \neg y_i$ können offensichtlich beide sehr schnell erzeugt werden. Beide Teilformeln bestehen dabei aus maximal $n = |Q|$ viele Variablen und $n - 1$ vielen logischen Operatoren.

Die rekursiv aufgebaute Teilformel $reach^A_n$ benötigt etwas mehr Aufwand. In jedem der n Rekursionsschritte beim Erzeugen der Formel werden $3 \cdot n$ viele Variablen quantifiziert. Außerdem werden sie durch $4 \cdot n$ Äquivalenzoperatoren ($\leftrightarrow$) miteinander in Beziehung gesetzt. Man sieht leicht, dass hierdurch nur ein polynomieller Mehraufwand entsteht.

Im letzten Rekursionsschritt wird die Teilformel $reach^A_0$ erzeugt. Auch diese Formel kann in polynomieller Zeit erzeugt werden. Für jeden Buchstaben $a \in \Sigma$ muss dabei die Formel $trans^A_a$ erstellt werden. Zu diesem Zweck wird im schlimmsten Fall für jeden Zustand $q_i \in Q$ jede Transition des Automaten besucht. Auch diese Teilformel kann also effizient erzeugt werden.

4.2 Inklusion

Gegeben seien zwei NEAs A und B mit Alphabet Σ, entscheide ob $L(B) \subseteq L(A)$. Prüfe zu diesem Zweck, ob $L(B) \cap \overline{L(A)} = \emptyset$.

Um Inklusion zu prüfen konstruiere wieder (siehe Kapitel 4.1) den Potenzautomaten A^d *on-the-fly* und suche ein Gegenbeispiel (d. h. $\exists w : \ w \in \mathcal{L}(B) \wedge w \notin \mathcal{L}(A)$).

Da für den Automaten B kein Komplement gebildet werden muss, wird für diesen Automaten die Potenzmengenkonstruktion nicht benötigt.

4.2.1 Notation

Sei $|Q_A| = n$ und $|Q_B| = m$.

Für die Kodierung einer Zustandsmenge M des Potenzautomaten A^d wird wie zuvor ein n-stelliges logisches Tupel verwendet (Siehe Kapitel 4.1.1).

Um die Formel möglichst klein zu halten, wird für die Zustände des NEA B eine binäre Kodierung verwendet [WTJK17]. Jeder Zustand $q_i \in Q_B = \{q_1, ..., q_m\}$ wird durch ein Tupel von $\lceil log_2(|Q_B|) \rceil$ vielen logischen Variablen x_i repräsentiert. Die Belegung der Variablen entsprechen der binären Repräsentation von i.

Falls über ein Tupel X von logischen Variablen gesprochen wird, dass einen Zustand binär kodieren soll, dann sei $\texttt{state}_\beta(X)$ definiert als: $\texttt{state}_\beta(X) = q_i$, mit $q_i \in Q_B$ und die Belegung der Variablen des Tupels X entspricht der binären Repräsentation von i. Andernfalls sei $\texttt{state}_\beta(X)$ für den Potenzautomaten A^d definiert wie zuvor. Der Ausdruck $\texttt{state}_\beta(X)$ wird zum Zweck der besseren Lesbarkeit also doppelt definiert. Aus dem Kontext, in dem diese Notation verwendet wird, ist immer ersichtlich welche Definition gemeint ist.

4.2.2 Konstruktion

Um auszudrücken, dass die Belegung für ein logisches Tupel X für den Automaten B einem bestimmten Zustand q_i entspricht, mit $b_1 b_2 \dots b_m$ als binärer Darstellung von i, sei die folgende Formel definiert:

$$\mathbf{bin}(q_i, X) = \bigwedge_{1 \leq j \leq m} \begin{cases} x_j & \text{wenn } b_j = 1 \\ \neg x_j & \text{sonst} \end{cases} \tag{4.8}$$

Um die binäre Repräsentation der Zustände des NEAs B zu berücksichtigen, muss die $trans_a$ Formel für den Automaten B etwas angepasst werden:

$$trans_a^B(X, Y) = \bigvee_{(q,p,a) \in \Delta_B} (\mathbf{bin}(q, X) \wedge \mathbf{bin}(p, Y)) \tag{4.9}$$

Für Inklusion müssen außerdem die neuen Formeln $trans_\Sigma^{B \subseteq A}$ und $reach_0^{B \subseteq A}$ konstruiert werden. Die Definition von $trans_a^A$ wird dabei aus Kapitel 4.1.2 übernommen. Durch die Formel $trans_\Sigma^{B \subseteq A}$ wird ausgedrückt, dass es in beiden Automaten einen Übergang zwischen den gewählten Zuständen unter dem gleichen Buchstaben gibt:

$$trans_\Sigma^{B \subseteq A}(X_A, Y_A, X_B, Y_B) = \bigvee_{a \in \Sigma} (trans_a^B(X_B, Y_B) \wedge trans_a^A(X_A, Y_A)) \tag{4.10}$$

Um auszudrücken, dass es sowohl in B als auch in A^d eine direkte Verbindung zwischen zwei gewählten Zuständen gibt, muss $reach_0^{B \subseteq A}$ wie folgt konstruiert werden:

$$reach_0^{B \subseteq A}(X_A, Y_A, X_B, Y_B) = (X_A \leftrightarrow Y_A \wedge X_B \leftrightarrow Y_B) \vee trans_\Sigma^{B \subseteq A}(X_A, Y_A, X_B, Y_B) \tag{4.11}$$

Die Formel $reach_i$ muss leicht angepasst werden, damit zwei Automaten gleichzeitig behandelt werden.

$$\begin{aligned} reach_i^{B \subseteq A}(X_A, Y_A, X_B, Y_B) = & \exists H_A, H_B \forall L_A, L_B \forall R_A, R_B \\ & \big((X_A \leftrightarrow L_A \wedge X_B \leftrightarrow L_B \wedge H_A \leftrightarrow R_A \wedge H_B \leftrightarrow R_B) \\ & \vee (H_A \leftrightarrow L_A \wedge H_B \leftrightarrow L_B \wedge Y_A \leftrightarrow R_A \wedge Y_B \leftrightarrow R_B)\big) \\ & \rightarrow reach_{i-1}^{B \subseteq A}(L_A, R_A, L_B, R_B) \end{aligned} \tag{4.12}$$

Ähnlich wie zuvor ergibt sich eine Beobachtung über die Länge von Gegenbeispielen:

Lemma 4.2.1. *Gegeben zwei NEAs A und B. $\forall w \in \mathcal{L}(B)$ und $w \notin \mathcal{L}(A)$ gilt:*
$\exists w' \in \mathcal{L}(B)$ und $w' \notin \mathcal{L}(A)$, sodass: $|w'| \leq |Q_B| \cdot 2^{|Q_A|}$

Beweis. Gegeben zwei NEAs A und B. Angenommen $\mathcal{L}(B) \not\subseteq \mathcal{L}(A)$. Sei $w \in \mathcal{L}(B)$ und $w \notin \mathcal{L}(A)$ und $|w| > |Q_B| \cdot 2^{|Q_A|}$ minimal. Sei $A^d = (P, \Sigma, p_1, \Delta_p, F_p)$ der Potenzautomat von A. q_1 sei der Anfangszustand des Automaten B und p_1 sei der Anfangszustand des Potenzautomaten A^d.

Dann gibt es zwei Zustände $q_B \in Q_B$ und $q_A \in P$, sodass $q_1 \overset{u}{\Longrightarrow}_B q_B \overset{k}{\Longrightarrow}_B q_B \overset{v}{\Longrightarrow}_B q'_B$ und $p_1 \overset{u}{\Longrightarrow}_A q_A \overset{k}{\Longrightarrow}_A q_A \overset{q'_A}{\Longrightarrow}_A$, wobei $w = ukv$, $k \neq \varepsilon$ und $q'_B \in F_B$ und $q'_A \notin F_p$. Da A^d deterministisch, gilt $w' = uv \notin \mathcal{L}(A) = \mathcal{L}(A^d)$. Offensichtlich gilt immer noch $w' \in \mathcal{L}(B)$. Also ist $|w|$ nicht minimal. □

Um Inklusion in QBF auszudrücken, wird erneut die negierte Formel konstruiert:

$$incl = \neg notincl \tag{4.13}$$

$$\begin{aligned} notincl = &\exists X_A, Y_A, X_B, Y_B \ \mathbf{bin}(q_{B,1}, X_B) \ \wedge \ initial_A(X_A) \\ &\wedge \bigvee_{q \in F_B} (\mathbf{bin}(q, Y_B)) \wedge \bigwedge_{q_i \in F_A} (\neg y_{A,i}) \\ &\wedge reach^{B \subseteq A}_{n \cdot \lceil log(m) \rceil}(X_A, Y_A, X_B, Y_B) \end{aligned} \tag{4.14}$$

4.2.3 Korrektheit

Auch für die Konstruktion zur Inklusion von NEAs wird die Korrektheit gezeigt. Zunächst werden wieder einige Eigenschaften der Teilformeln benötigt:

Lemma 4.2.2. *Gegeben zwei NEAs A und B, sowie eine Belegung β gilt:*
$\beta \models trans^B_a(X, Y)$ gdw. $\texttt{state}_\beta(X) \xrightarrow{a}_B \texttt{state}_\beta(Y)$

Beweis. Wenn es eine Transition von $\texttt{state}_\beta(X)$ nach $\texttt{state}_\beta(Y)$ unter a gibt, wertet die Formel offensichtlich zu `true` aus. Andernfalls wird keine der Konjunkte erfüllt. □

Analog zu Lemma 4.1.3 und Lemma 4.1.4 folgen die nächsten beiden Aussagen.

Korollar 4.2.3. *Gegeben zwei NEAs A und B, sowie eine Belegung β gilt:*
$\beta \models trans^{B \subseteq A}_\Sigma(X_A, Y_A, X_B, Y_B)$
gdw. $\exists a \in \Sigma : \texttt{state}_\beta(X_A) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y_A)$ und $\texttt{state}_\beta(X_B) \xrightarrow{a}_B \texttt{state}_\beta(Y_B)$

Korollar 4.2.4. *Gegeben zwei NEAs A und B, sowie eine Belegung β gilt:*
$\beta \models reach^{B \subseteq A}_0(X_A, Y_A, X_B, Y_B)$
gdw. $(\exists a \in \Sigma : \texttt{state}_\beta(X_A) \xrightarrow{a}_{A^d} \texttt{state}_\beta(Y_A)$ und $\texttt{state}_\beta(X_B) \xrightarrow{a}_B \texttt{state}_\beta(Y_B))$
oder $(\texttt{state}_\beta(X_A) = \texttt{state}_\beta(Y_A)$ und $\texttt{state}_\beta(X_B) = \texttt{state}_\beta(Y_B))$

Auch die *reach*-Formel ist ähnlich aufgebaut wie in Lemma 4.1.5.

Lemma 4.2.5. *Gegeben zwei NEAs A und B, sowie eine Belegung β gilt:*
$\beta \models reach^{B \subseteq A}_i(X_A, Y_A, X_B, Y_B)$ gdw. $\exists w \in \Sigma^* : |w| \leq 2^i$ und
$\texttt{state}_\beta(X_A) \overset{w}{\Longrightarrow}_{A^d} \texttt{state}_\beta(Y_A)$ und $\texttt{state}_\beta(X_B) \overset{w}{\Longrightarrow}_B \texttt{state}_\beta(Y_B)$

Beweis. Ähnlich wie in Lemma 4.1.5 gibt es zwei Möglichkeiten L und R zu belegen, sodass die Formel $reach^{B \subseteq A}_{i-1}$ erfüllt sein muss. Dadurch teilt sich der Pfad in zwei Teile, für die Gegenbeispiele der Länge $\leq 2^{i-1}$ untersucht werden. Analog zu Lemma 4.1.5 folgt die zu zeigende Aussage aus der Induktion über i. □

Jetzt lässt sich ähnlich wie im vorherigen Kapitel das Hauptresultat für Inklusion zeigen:

Theorem 4.2.6. *notincl ist gültig gdw.* $\mathcal{L}(B) \not\subseteq \mathcal{L}(A)$

Beweis. Der erste Teil der Formel stellt sicher, dass X_A und X_B die Startzustände der Automaten A^d und B darstellen. Aus dem zweiten Teil ergibt sich, dass $\texttt{state}_\beta(Y_B)$ ein Endzustand von B ist. Als nächstes wird, ähnlich zur Formel für Universalität (Theorem 4.1.7), sichergestellt, dass $\texttt{state}_\beta(Y_A)$ keine Endzustände des Eingabeautomaten A enthält. Nach Lemma 4.2.5 erfüllt eine Belegung β die Teilformel $reach^{B\subseteq A}_{n\cdot\lceil log(m)\rceil}$ genau dann, wenn es einen entsprechenden Pfad in beiden Automaten gibt. (d. h. $\exists\beta: \ \beta \models reach^{B\subseteq A}_{n\cdot\lceil log(m)\rceil}$) Also gibt es genau dann ein Gegenbeispiel w mit $|w| \leq n \cdot \lceil log(m)\rceil$, wenn die Formel *notincl* erfüllbar ist.

Nach Lemma 4.2.1 gibt es auch nur genau dann ein Gegenbeispiel w von beliebiger Länge. □

Platzverbrauch der Formel

Ähnlich wie bereits für Universalität am Ende des Kapitels 4.1.3 beschrieben, können auch die QBFs für Inklusion effizient generiert werden. Da für Inklusion längere Gegenbeispiele betrachtet werden müssen, sind die generierten Formeln zwar größer, es werden jedoch weiterhin nur polynomiell große Formeln erzeugt.

4.3 Optimierung der Kodierung

Die generierten Formeln nehmen schnell an Größe zu und sind für QBF-Solver nur schwer zu lösen (Siehe Kapitel 6). Es liegt nahe, nach Ansätzen zu suchen, die die Formeln vereinfachen bzw. umstrukturieren, damit diese für einen QBF-Solver leichter zu lösen sind.

Da die Formeln selbst sehr einfach aufgebaut sind, ist es schwierig alternative Kodierungen zu finden. Dementsprechend ist es sinnvoll, die Eingabeautomaten anzupassen und deren Aufbau in der Kodierung der Formel auszunutzen.

4.3.1 Vereinfachung der Endzustände

Ein Solver, der die ursprünglich vorgestellte QBF lösen soll, muss die logischen Variablen des Tupels Y belegen. Dabei gibt es kaum Einschränkungen, da zu Beginn sichergestellt werden muss, dass $\texttt{state}_\beta(Y)$ keinen Endzustand von A enthält. Um diesen Suchraum zu verkleinern, bietet sich eine Anpassung des Eingabeautomaten A zu A' an. Die Idee ist es, den Automaten so zu verändern, dass jede Berechnung in einem von zwei Zuständen enden muss. Dadurch kann der äußere Teil der Formel etwas vereinfacht werden, denn die Berechnung für ein Gegenbeispiel muss nun in der Zustandsmenge $\{q_rej\}$ enden. Hier beispielhaft für Universalität demonstriert. Der Ansatz lässt sich leicht auf Inklusion erweitern.

$$notuniv = \exists X \exists Y \ initial^A(X) \wedge (\bigwedge_{q_i \in F} \neg y_i) \wedge reach^A_n(X, Y)$$

Wird zu:

$$notuniv = \exists X \exists Y \ initial^{A'}(X) \wedge y_{rej} \wedge \bigwedge_{\substack{q_i \in Q' \\ q_i \neq q_{rej}}} (\neg y_i) \wedge reach^{A'}_{n+1}(X, Y)$$

Wobei q_{rej} und q_{acc} zwei neue Zustände des Automaten A' sind.
Gesucht ist ein Automat A', sodass gilt: $w \in L(A)$ *gdw.* $wx \in L(A')$, mit $x \notin \Sigma$.

Erweitere zu diesem Zweck das Alphabet Σ um einen weiteren Buchstaben x. Außerdem füge zwei neue Zustände q_{acc} und q_{rej} ein. Für jeden Zustand $q \in Q \setminus F$ führe eine zusätzliche Transition (q, x, q_{rej}) ein, sowie für jeden Zustand $q \in F$ analog die Transition (q, x, q_{acc}). Die neue Menge der Endzustände ist $F' = \{q_{acc}\}$. Die Abbildung 4.1 skizziert diese Konstruktion. Die Startzustände wurden weggelassen.

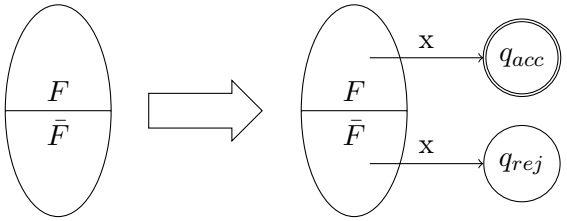


Abbildung 4.1: Modifizierter endlicher Automat mit verwerfenden und akzeptierenden Zustand

Für alle $w \in \mathcal{L}(A)$ gilt, dass $q_1 \overset{w}{\Longrightarrow}_{A'} p \overset{x}{\rightarrow}_{A'} q_{acc}$ für ein $p \in F$. Außerdem gilt für alle $wx \in \mathcal{L}(A')$, dass $q_1 \overset{wx}{\Longrightarrow}_{A'} q_{acc}$. Nach Konstruktion folgt also: $q_1 \overset{w}{\Longrightarrow}_{A} p \overset{x}{\rightarrow}_{A'} q_{acc}$ für ein $p \in F$. Analog endet für jedes Wort $w \in \Sigma^*$, das nicht von A akzeptiert wird, die Berechnung für das Wort $wx \in (\Sigma \cup \{x\})^*$ in A' im Zustand q_{rej}.

Die letzte Eigenschaft gilt allerdings nur für Berechnungen, die in einem Zustand enden und nicht aufgrund fehlender Transitionen vorher abgebrochen werden. Im Potenzautomaten enden solche Berechnungen in der Zustandsmenge $\emptyset$. Damit jede Berechnung im modifizierten Potenzautomat A'^d entweder in der Zustandsmenge $\{q_{acc}\}$ oder $\{q_{rej}\}$ endet, muss der Zustand $\emptyset$ des Potenzautomaten ausgeschlossen werden. Dies ist nur möglich, wenn der Eingabeautomat für jeden Buchstaben und Zustand mindestens einen Übergang hat. Das bedeutet, dass der Eingabeautomat eventuell um einen Zustand erweitert werden muss.

4.3.2 Umstrukturierung der Formel

Die oben vorgestellte Variante besitzt eine Eigenschaft, die es ermöglicht, die Formel weiter zu vereinfachen. Jede Berechnung im Automaten A' endet entweder im Zustand q_{acc} oder q_{rej}. Da kein Übergang mit dem Buchstaben x existiert, der in einen anderen Zustand führt, gibt es insbesondere keine Berechnungen der Form $q_1 \overset{u}{\Longrightarrow}_{A'} q \overset{x}{\rightarrow}_{A'} q' \overset{v}{\Longrightarrow}_{A'} p$.

Alle relevanten Berechnungen (Berechnungen für Wörter der Form wx) verwenden genau einmal einen Übergang mit dem Buchstaben x. Da diese Transition immer der letzte Übergang der Berechnung sein muss, ist es möglich diesen Übergang in der Formel separat zu behandeln.

$$notuniv = \exists X \exists Y \; initial^{A'}(X) \wedge y_{rej} \wedge \bigwedge_{\substack{q_i \in Q' \\ q_i \neq q_{rej}}} (\neg y_i) \wedge reach_{n+1}^{A'}(X, Y)$$

Wird zu:

$$notuniv = \exists X \exists Y \exists Z \; initial^{A'}(X) \wedge \bigwedge_{\substack{q_i \in Q \\ q_i \neq q_{rej}}} (\neg z_i) \wedge reach_{n}^{A}(X, Y) \wedge trans_{x}^{A'}(Y, Z)$$

Da die neuen beiden Zustände nur im letzten Übergang vorkommen können, müssen sie während der Suche der vorgehenden Übergänge nicht betrachtet werden. Demnach sinkt die Iterationstiefe bei der Generierung der Formel.

Gesucht ist also ein Pfad $\texttt{state}_\beta(X) \overset{w}{\Longrightarrow}_A \texttt{state}_\beta(Y) \overset{x}{\rightarrow}_{A'} \texttt{state}_\beta(Z)$.

Da $w \in \Sigma^*$ müssen die in Kapitel 4.3.1 eingeführten Übergänge innerhalb der *reach*-Formel ebenfalls nicht beachtet werden. Für die Konstruktion der $trans_\Sigma$-Formel muss also nur noch das Alphabet des Automaten A verwendet werden. Dadurch sinkt die Anzahl von Disjunktionen (im Vergleich zu Variante 1) innerhalb der Teilformel $trans_\Sigma$ wieder. Ein Solver hat also wieder weniger Möglichkeiten die logischen Variablen zu belegen.

Da jetzt eigentlich nur noch im ursprünglichen Automaten ein Pfad gesucht wird und die beiden neuen Zustände innerhalb der *reach*-Formel nicht mehr betrachtet werden müssen, muss bei dieser Variante nicht mehr gefordert werden, dass es im Eingabeautomaten mindestens einen Übergang für jeden Buchstaben und Zustand gibt.

Durch die Einführung des Tupel logischer Variablen Y gibt es allerdings erneut ein Tupel, welches vom QBF-Solver belegt werden muss. Die QBF ist also im Vergleich zur ursprünglichen Kodierung tendenziellsy komplexer geworden. Da diese Variante nur eine weitere Vereinfachung der Vorherigen ist, wird vermutlich keine der beiden Varianten für einen QBF-Solver einfacher zu lösen sein. (Siehe Kapitel 6.3)

4.3.3 Einschränken der Formel auf Gegenbeispiele der Länge 2^n

Die Formel $reach_0$ verwendet einige logische Operatoren, um Gegenbeispiele der Länge $|w| < 2^n$ zu finden. Durch eine Modifikation des in Kapitel 4.3.1 eingeführten Automaten A' genügt es, Gegenbeispiele der Länge exakt $|w| = 2^n$ zu suchen.

Erweitere zu diesem Zweck den Automaten A' zu A'', indem für die Zustände q_{rej} und q_{acc} die Transitionen (q_{rej}, x, q_{rej}) und (q_{acc}, x, q_{acc}) eingeführt werden. (Siehe Abbildung 4.2)

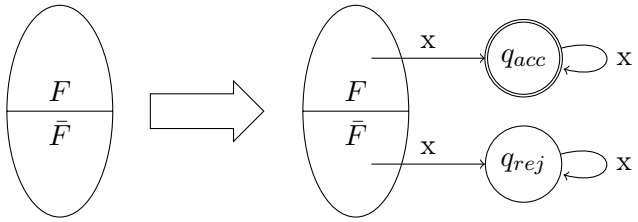


Abbildung 4.2: Modifizierter endlicher Automat für Wörter der Länge 2^n

Offensichtlich ist $\mathcal{L}(A'') = \{uv \mid u \in \mathcal{L}(A) \text{ und } v \in \{x\}^*, |v| > 0\}$.
D. h. $\mathcal{L}(A) \neq \Sigma^*$ *gdw.* $\exists w \in \Sigma^* : w \notin \mathcal{L}(A)$ *gdw.* $uv \notin \mathcal{L}(A'')$ mit $u = w \wedge v \in \{x\}^*$.
Die grundlegende Formel aus Kapitel 4.3.1 muss nicht verändert werden.

$$notuniv = \exists X \exists Y \; initial^{A''}(X) \wedge y_{rej} \wedge \bigwedge_{\substack{q_i \in Q' \\ q_i \neq q_{rej}}} (\neg y_i) \wedge reach_{n+1}^{A''}(X, Y)$$

Allerdings kann die Formel $reach_0$ wie folgt verkleinert werden:

$$reach_0^A(X, Y) = (X \leftrightarrow Y) \vee trans_\Sigma^A(X, Y)$$

Wird zu:

$$reach_0^{A''}(X, Y) = trans_\Sigma^{A''}(X, Y)$$

Ähnlich zur ersten Variante (Kapitel 4.3.1) wird als Eingabe wieder ein Automat gefordert, der für jeden Buchstaben und Zustand mindestens eine Transition enthält.

4.3.4 Kurze Gegenbeispiele

In Kapitel 6.1 wird festgestellt, dass einige der Formeln (der ursprünglichen Kodierung) sehr schnell gelöst werden. Dies wurde auf die leere Zustandsmenge zurückgeführt. Wenn dieser Zustand im Potenzautomaten A^d erreicht werden kann, ist ein QBF-Solver in der Lage zu erkennen, dass alle Folgezustände im Potenzautomaten ebenfalls nur der leeren Zustandsmenge entsprechen können.

Um den Suchraum auch für andere Gegenbeispiele zu verkleinern, kann die Formel aufgrund folgender Beobachtung leicht angepasst werden. Diese Variante kann also das feststellen von Universalität (oder Inklusion) nicht beschleunigen, sondern lediglich das Identifizieren von nicht universellen Automaten (bzw. nicht enthaltenen Automaten) erleichtern.

Die $reach(X, Y)$-Formel trennt den Pfad eines potentiellen Gegenbeispiels *mittig* in zwei Teile. Zu diesem Zweck wird eine Belegung für H gewählt.

Beobachtung 4.3.1. *Für eine Belegung β und $w = uv$, mit $\texttt{state}_\beta(X) \overset{u}{\Longrightarrow}_A \texttt{state}_\beta(H) \overset{v}{\Longrightarrow}_A \texttt{state}_\beta(Y)$. Wenn $\forall q \in \texttt{state}_\beta(H) : \; q \notin F$, dann ist u ein Gegenbeispiel.*

Wenn H also bereits nicht einem Endzustand entspricht, würde mit dem ersten Teil des Pfades bereits ein Gegenbeispiel gefunden werden. In diesem Fall muss der zweite Teil des Pfades ($\texttt{state}_\beta(H) \overset{v}{\Longrightarrow}_A \texttt{state}_\beta(Y)$) nicht mehr überprüft werden. Die $reach(X, Y)$-Formel kann also wie folgt verändert werden:

$$reach_i^A(X, Y) = \exists M \forall L \forall R \big((X \leftrightarrow L \wedge M \leftrightarrow R) \vee (M \leftrightarrow L \wedge Y \leftrightarrow R) \big) \rightarrow reach_{i-1}^A(L, R)$$

Wird zu:

$$\begin{aligned} reach_i^A(X, Y) = & \exists M \forall L \forall R \big((X \leftrightarrow L \wedge M \leftrightarrow R) \vee (M \leftrightarrow L \wedge Y \leftrightarrow R \wedge \bigvee_{q_i \in F} m_i) \big) \\ & \rightarrow reach_{i-1}^A(L, R) \end{aligned}$$

Auch diese Variante lässt sich leicht für die Formeln für Inklusion erweitern. Außerdem kann diese Modifikation mit einer der anderen drei vorgestellten Varianten kombiniert werden.

4.3.5 Zusammenfassung der Varianten

Es wurden vier verschiedene Anpassungen der Formeln und Eingabeautomaten vorgestellt.

Die in Kapitel 4.3.1 eingeführte Variante benötigt einen größeren Automaten und verwendet dementsprechend eine größere Formel. Dabei werden sowohl durch eingeführte Zustände (bis zu drei), als auch durch neue Transitionen ($> |Q|$) mehr logische Operatoren benötigt.

Die in Kapitel 4.3.2 eingeführte Variante, in der die neuen Zustände in der Formel getrennt behandelt werden, kann im Vergleich zur ersten Variante einige Operatoren einsparen. Da für diese Kodierung der Eingabeautomat immer nur um zwei Zustände erweitert wird, kann verhindert werden, dass die Formel sich unter Umständen noch weiter vergrößert. Diese Variante verwendet dennoch mehr logische Operatoren als die ursprüngliche Kodierung (Siehe Kapitel 6.3).

Die in Kapitel 4.3.3 eingeführte Variante, die nur Gegenbeispiele der Länge 2^n sucht, hat ähnliche Einschränkungen, wie die erste Variante.

In Kapitel 4.3.4 wurde eine Variante vorgestellt, die das Finden von kurzen Gegenbeispielen erleichtern soll. Für diese Variante werden allerdings keine Modifikationen am Eingabeautomaten benötigt.

Durch die drei zu Beginn vorgestellten Varianten werden zusätzliche Zustände im Automaten eingeführt, welche in der Formel berücksichtigt werden. Alle vier Varianten verwenden größere Formeln als die ursprüngliche Kodierung (Kapitel 6.2). Da gerade die ersten beiden Varianten sich nur wenig von der ursprünglichen Kodierung unterscheiden und durch den größeren Eingabeautomaten komplexer sind, können hier nicht unbedingt Verbesserungen in der Performanz erwartet werden. Die beobachtete Performanz wird in Kapitel 6.3 erläutert.

5 Implementation

Um die in Kapitel 4 beschriebenen Kodierungen für Universalität und Inklusion zu nutzen, wurde eine Software in der JVM-basierten Sprache Kotlin implementiert.[1] Die Implementierung realisiert die Umwandlung der Automaten in eine QBF. Um die Entscheidungsprobleme zu lösen, muss also zusätzlich ein beliebiger QBF-Solver verwendet werden.

5.1 Kompilierung

Der Quellcode der Software ist im Anhang dieser Arbeit enthalten. Außerdem ist er verfügbar unter: `http://brenig.net/downloads/jonas/nfa-to-qbf.zip`

Für die Kompilierung wird das Buildsystem `Gradle`[2] verwendet. Vorausgesetzt das Java Development Kit[3] (Version 8 oder höher) ist verfügbar, kann die Software durch `./gradlew` (bzw. `gradlew.bat`) kompiliert werden.

5.2 Verwendung

Für die Ausführung des Programms wird die Java-Laufzeitumgebung (Version 8 oder höher) benötigt. Als Eingabe werden ein bzw. zwei Automaten im `.ba`-Format[4] als Parameter erwartet.

Die Ausgabe wird im, von vielen Solvern verwendete, `.qcir`-Format[5] in eine Datei geschrieben. Optional kann, mit dem Parameter `-o`, der Name der Ausgabedatei übergeben werden.

Mit den Optionen `-u`, `-i` und `-e` kann angegeben werden, ob eine Formel für Universalität, Inklusion oder Äquivalenz erzeugt werden soll. Weitere Optionen können mit dem Argument `-h` aufgezählt werden.

Ein Beispielaufruf sieht also aus wie folgt:

```
java -jar ./nfa-to-qbf.jar ./input1.ba ./input2.ba -o ./outputfile.qcir -i
```

Die auf diese Weise generierte Formel kann anschließend von einem QBF-Solver verwendet werden, um das entsprechende Entscheidungsproblem zu entscheiden. Der verwendete Solver muss dafür Formeln im `.qcir`-Format verarbeiten können.

Das in Kapitel 3.4 vorgestellte Verfahren zur Minimierung von Automaten von Mayr und Clemente [MC13] verwendet das gleiche `.ba`-Eingabeformat und kann ohne weiteres als Vorverarbeitungsschritt benutzt werden.

Ebenfalls ist es möglich den Eingabeautomaten mithilfe des Parameters `-c` in das `.gff` Format umzuwandeln, welches von dem Tool `GOAL` [TCT+08] verwendet wird. Dies ist nützlich, um die Geschwindigkeit der beiden Ansätze zu vergleichen.

[1] Kotlin programming language: `https://kotlinlang.org/`
[2] Gradle build system: `https://gradle.org/`
[3] JDK Downloads: `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`
[4] `.ba`-Format: `http://www.languageinclusion.org/doku.php?id=tools`
[5] `.qcir`-Format: `http://qbf.satisfiability.org/gallery/qcir-gallery14.pdf`

5.3 Implementationsdetails

Die Software ist in mehreren *packages* strukturiert. Unter `net.brenig.nfa.automaton` ist die Repräsentation von endlichen Automaten, sowie Im- und Export von endlichen Automaten im `.ba`-Format (Abbildung 5.1) zu finden. Außerdem sind dort die verwendeten Vorverarbeitungsschritte für Automaten enthalten. (Siehe Kapitel 4.3.1)

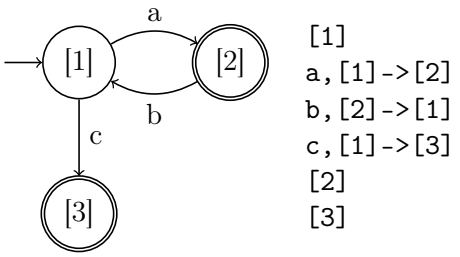


Abbildung 5.1: Beispiel des `.ba`-Formats (Siehe `languageinclusion.org`)

Unter `net.brenig.nfa.qbf` ist die Repräsentation von logischen Formeln, sowie Im- und Export von logischen Formeln im `.qcir`-Format zu finden.

Die eigentlichen Kodierungen sind im *package* `net.brenig.nfa.reduction` abgelegt. Die Implementation für die Kodierungen zur Universalität sind dabei in der Datei `Universality.kt` zu finden. In der Datei `Inclusion.kt` ist die Implementation für die Kodierungen zur Inklusion enthalten.

Da einige der Formeln für beide Probleme verwendet werden, sind einige verwendete Teilformeln in der Datei `Common.kt` abgelegt. Die Klasse `net.brenig.nfa.Settings` beinhaltet alle Parameter für die Formelerstellung (Siehe Kapitel 4.3).

5.4 Erzeugung der QBF

Für die Erzeugung von logischen Formeln sind vor allem die Typen `QBFBuilder` und `Builder` verantwortlich.

Die Klasse `QBFBuilder` beinhaltet Hilfsfunktionen für verschiedene logische Operatoren. Um die Formel direkt in Pränexnormalform zu generieren, verwaltet die Klasse `QBFBuilder` außerdem die Quantoren der Formel.

Um während der Generierung einer Teilformel zu wissen, welcher Quantor an die Quantorenliste angehängt werden muss, wird gespeichert wie oft die aktuelle Formel negiert wurde. Da $\neg\exists x\varphi$ äquivalent ist zu $\forall x\neg\varphi$ wird hierfür ein Wahrheitswert benötigt. Für jede Negation der Teilformel wird dieser Wahrheitswert negiert. Wenn innerhalb einer `Builder`-Funktion also quantifiziert wird, wird direkt der richtige Quantor ($\forall$ oder $\exists$) gewählt und an die Quantorenliste angehängt.

Zum generieren einer Teilformel wird der Typ `Builder` verwendet. Dieser repräsentiert eine beliebige Funktion, welche einen `Boolean`-Wert annimmt und für die Konstruktion einer logischen Formel (`BooleanFormula`) verantwortlich ist. In dem übergebenen `Boolean`-Parameter wird festgehalten, wie oft die zu konstruierende Formel bereits negiert wurde.

Eine Negation (siehe `QBFBuilder::negate`) erwartet z. B. zwei Parameter. Zum einen den aktuellen *Negationsstatus* der Formel als Wahrheitswert (siehe oben) und als zweites Argument einen `Builder` für die Formel die negiert werden soll. Die übergebene `Builder`-Funktion wird dann mit dem Inversen des aktuellen *Negationsstatus* ausgeführt (da es sich um den Negationsoperator handelt) und die negierte Formel zurückgegeben (siehe `net.brenig.nfa.qbf.Negation`).

6 Experimentelle Auswertung

In diesem Kapitel soll die Effektivität des vorgestellten Ansatzes analysiert werden. Dabei soll einerseits festgestellt werden, wie viel Zeit das Lösen und Erzeugen der QBFs in Anspruch nimmt und wie sich diese Ergebnisse mit der Performanz anderer Algorithmen vergleichen. Anschließend werden in Kapitel 6.2 die generierten Formeln bezüglich der Anzahl von Operatoren, Quantorenalternierungen und *treewidth* untersucht, um mögliche Verbesserungen der Kodierung zu identifizieren.

Als Basis für die experimentelle Auswertung dienen zufällig generierte Automaten, basierend auf dem Modell von Tabakov und Vardi [TV05]. Die verwendete Implementierung ist Teil des von Mayr und Clemente entwickelten Werkzeugs RABIT (Siehe Kapitel 3.4). RABIT kann außerdem dazu verwendet werden, um die Eingabeautomaten zu verkleinern. Zunächst wird auf diesen Schritt verzichtet. Am Ende von Kapitel 6.1 wird diskutiert, welche Verbesserungen der benötigten Zeit durch eine Vorverarbeitung der Automaten durch RABIT möglich sind.

Alle Experimente wurden auf einem `Intel i5 2500k @4.3GhZ` Prozessor mit `16Gb` Arbeitsspeicher durchgeführt. Für alle Experimente wurde ein Timeout von 30 Minuten verwendet.

Für das Lösen von QBFs gibt es eine große Anzahl an Solvern. Jedes Jahr treten Solver in der QBFEval[1] gegeneinander an. Unterschieden wird dabei zwischen Solvern, welche Formeln in Pränexnormalform lösen können, und Solvern, die Formeln in konjunktiver Normalform benötigen. Die für Universalität und Inklusion erzeugten Formeln liegen lediglich in Pränexnormalform vor. Daher bieten sich Solver an, die das `.qcir` direkt verwenden können.

Für die Auswertung wurde der hauptsächlich Solver `QuAbs`[2] von Leander Tentrup ausgewählt [Ten16]. Dieser Solver hat 2017 bei der QBFEval den dritten Platz belegt und in, für diese Arbeit durchgeführten, Tests die besten Resultate für die erzeugten Formeln geliefert. (Siehe Abbildung 6.5)

6.1 Lösen der Formeln

Das Erzeugen der Formeln, sowie die Umwandlung in konjunktive Normalform (KNF) (falls nötig), benötigt nur sehr wenig Zeit. Selbst für sehr große Automaten mit 1000 Zuständen wird für die Generierung der Formeln für Universalität unter 500ms benötigt. Auch eine Konvertierung in das KNF-Format `.qdiamcs` benötigt ebenfalls wenig Zeit. Da die Formeln für Inklusion wesentlich größer sind, wird hier etwas mehr Zeit benötigt. Bei 100 Zuständen wird für die Generierung der Formeln für Inklusion trotzdem weniger als eine Sekunde benötigt.

Ausschlaggebend ist also die Zeit, die benötigt wird, um die generierte QBF zu lösen. Im weiteren Verlauf des Kapitels wird nur dieser Wert betrachtet.

Für jede Automatengröße wurden jeweils 10 Automaten generiert. Für die Generierung der Automaten werden die Parameter *transition density* und *accepting state density* benötigt, welche die Anzahl an Transitionen und Endzuständen beeinflussen. Dabei wurden die Parameter *transition density* auf 2.0 und *accepting state density* auf 0.5 gesetzt. Die

[1] QBFEval: `http://www.qbflib.org/index_eval.php`
[2] QuAbs: `https://www.react.uni-saarland.de/tools/quabs/`

Werte für diese Parameter orientieren sich an dem verwendeten Wertebereich der Parameter in der Arbeit von Mayr und Clemente, in der das Programm RABIT vorgestellt wurde [TV05]. In Abbildung 6.6 werden die Auswirkungen der Wahl der Parameter auf die benötigte Zeit zum Lösen der Formel dargestellt.

Universalität

Die Zeit, die ein QBF-Solver benötigt, um Universalität zu lösen, steigt bereits bei einer sehr geringen Anzahl von Zuständen stark an. In Abbildung 6.1 ist dieser starke Anstieg der durchschnittlichen benötigten Zeit bei etwa 7 Zuständen zu erkennen. Die Fehlerbalken geben dabei die Standardabweichung an.

Im Vergleich zu existierenden Implementationen anderer Techniken, wie z. B. GOAL und RABIT (Siehe Kapitel 3.1/3.4), ist dies ein sehr schlechtes Resultat. Selbst bei Automaten mit 100 Zuständen können sowohl GOAL als auch RABIT Universalität und auch Inklusion in weniger als 5 Sekunden entscheiden. Diese Verfahren können wesentlich größere Automaten mit über 1000 Zuständen verarbeiten.

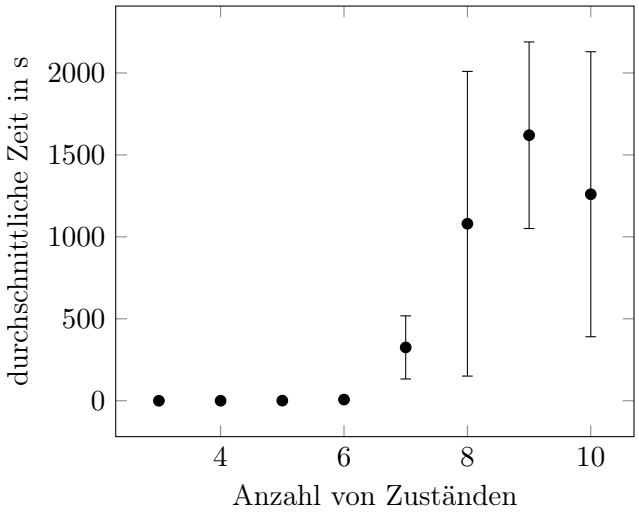


Abbildung 6.1: benötigte Zeit zum Lösen von Universalität (QuAbs)

Für die vom Solver benötigte Zeit gibt es außerdem eine hohe Standardabweichung, da einige Instanzen in sehr kurzer Zeit entschieden werden, während andere Formeln für Automaten der gleichen Größe nicht innerhalb des Timeouts von 30 Minuten gelöst werden.

Wie in Abbildung 6.2 zu sehen, kann die Formel für Universalität von QuAbs schon ab Automaten mit 8 Zuständen nur noch für 4 der 10 Automaten innerhalb von 30 Minuten gelöst werden. Obwohl QuAbs bei Automaten mit 8 Zuständen nur 4 Formeln lösen konnte, wurde jede dieser Formeln in unter einer Sekunde entschieden.

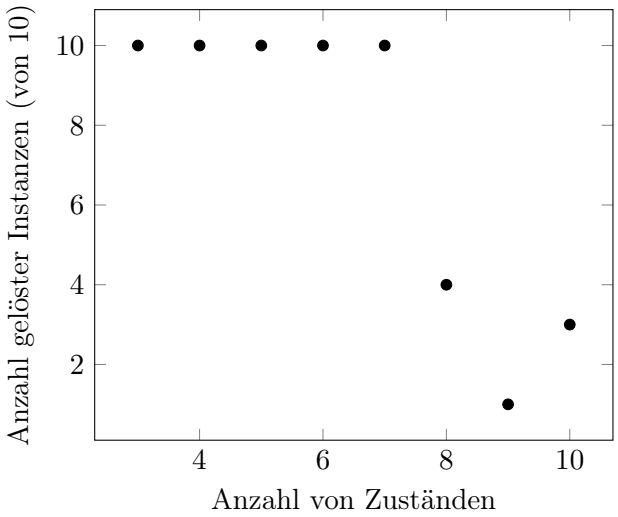


Abbildung 6.2: Gelöste Instanzen (QuAbs)

Durch eine leichte Anpassung verschwindet dieser Effekt. Der Solver benötigt für diese Formeln dann allerdings wesentlich mehr Zeit (Abbildung 6.3). Für Automaten mit 7 Zuständen ist zwar eine hohe Standardabweichung zu erkennen, allerdings wurde keine der 3 gelösten Instanzen in weniger als 290 Sekunden gelöst. Modifiziert man den Eingabeautomaten, sodass dieser in jedem Zustand mindestens einen Übergang für jeden Buchstaben (siehe Kapitel 4.3.1) besitzt, kann die Formel entsprechend verändert werden, damit die leere Zustandsmenge im Potenzautomaten ausgeschlossen wird. Da der Effekt nur bei nicht universellen Automaten auftritt, ist bei den zugehörigen Potenzautomaten vermutlich die leere Zustandsmenge erreichbar.

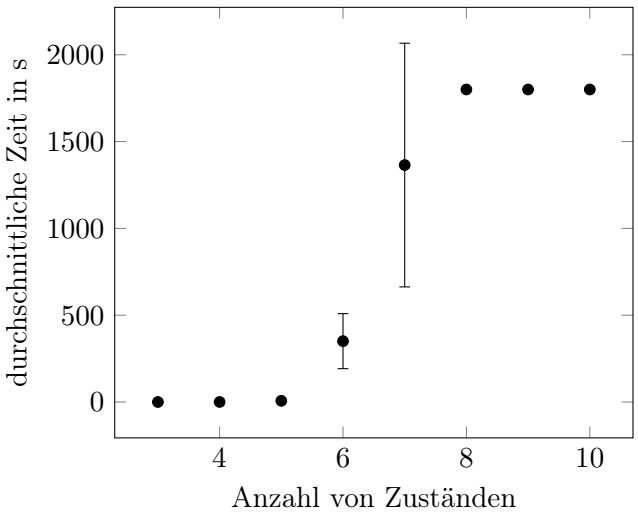


Abbildung 6.3: benötigte Zeit zum Lösen der modifizierten Formel (QuAbs)

Inklusion

Das Entscheiden von Inklusion mithilfe von QBF-Solvern ist langsamer. Da pro Formel für Inklusion jeweils zwei Eingabeautomaten benötigt werden, wurden dieses Mal 20 zufällige Automaten generiert. Für je zwei dieser Automaten wurde eine QBF erzeugt, sodass wieder 10 Formeln für jede Automatengröße getestet wurden. Der Solver QuAbs benötigt hier schon bei Automaten mit jeweils 4 Zuständen sehr viel Zeit und kann die Formeln oft nur in knapp innerhalb dem gewählten Timeout von 30 Minuten lösen.

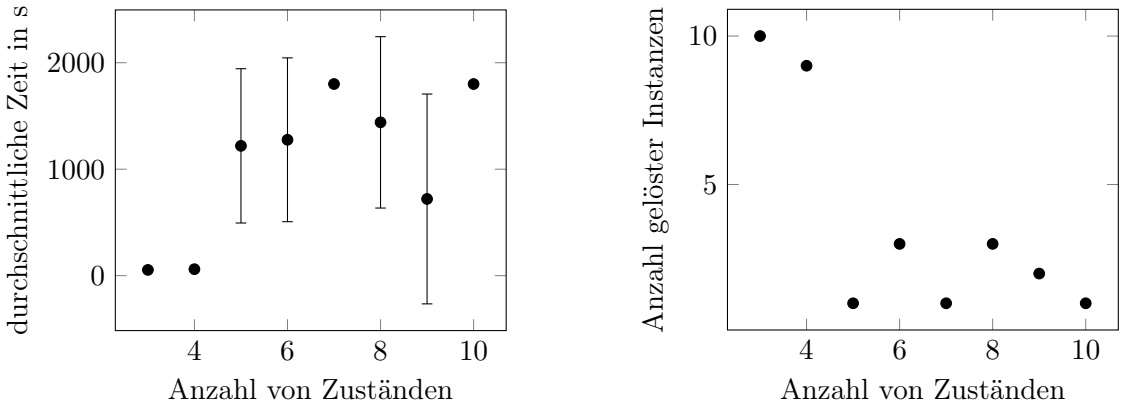


Abbildung 6.4: benötigte Zeit zum Lösen von Inklusion (QuAbs)

Solver im Vergleich

Wie bereits erwähnt, schneidet QuAbs im Vergleich mit anderen Solvern am besten ab. In Abbildung 6.5 wird die Geschwindigkeit einiger Solver für das Lösen von Universalität verglichen. Die Auswahl der Solver wurde dabei auf Basis des Abschneidens in der QBFEval 2017 und der Zugänglichkeit der Software getroffen.

Der Solver GhostQ[3] kann bereits bei 7 Zuständen keine der Formeln mehr lösen. Der KNF-Solver DepQBF[4] erreicht schon bei Automaten mit 5 Zuständen das Timeout von 30 Minuten. Da die Formel für den Solver DepQBF automatisiert in KNF konvertiert wurde, ist hier durch eine manuelle Kodierung der Formel in KNF möglicherweise eine Verbesserung dieser Ergebnisse möglich.

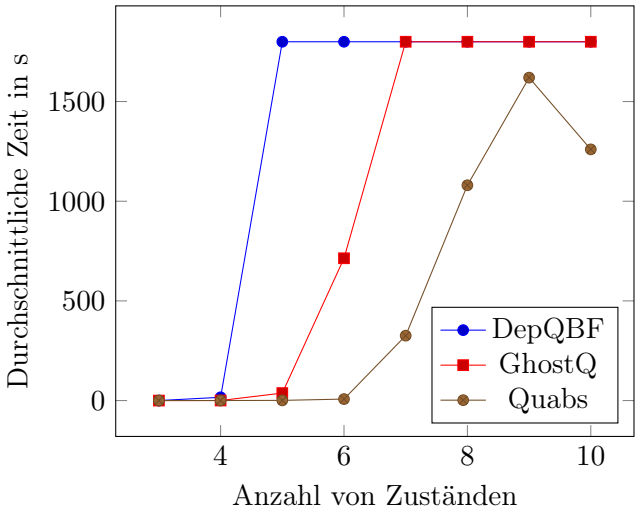


Abbildung 6.5: QBF-Solver im Vergleich

Parameter der Zufallsgenerierung der Automaten

Für die Generierung der Automaten wird eine Implementierung auf Basis des Modells von Tabakov und Vardi verwendet [TV05]. Für dieses Werkzeug werden die Parameter `transition density` (td), der die Anzahl an Transitionen im Automaten beeinflusst, und `accepting state density` (ad), der die Anzahl von Endzuständen beeinflusst, benötigt.

Die Parameter, mit denen die Automaten generiert werden, haben einen direkten Einfluss darauf, wie schnell der QBF-Solver die Formeln lösen kann. Eine höhere `transition`

[3] GhostQ: `https://www.cs.cmu.edu/~wklieber/ghostq/`
[4] DepQBF: `http://lonsing.github.io/depqbf/`

`density` führt zu Automaten, für die Universalität mittels QBF-Solvern schwerer zu entscheiden ist (Siehe Abbildung 6.6). Bei einer geringeren `transition density` erhöht sich die Anzahl an Automaten, für die die generierten Formeln innerhalb des Timeouts gelöst werden können. Der Parameter `accepting state density` scheint keinen großen Einfluss auf die benötigte Zeit zu haben. Diese Beobachtungen treffen auch auf die von RABIT benötigte Zeit zu (getestet bei Automaten mit 100 Zuständen).

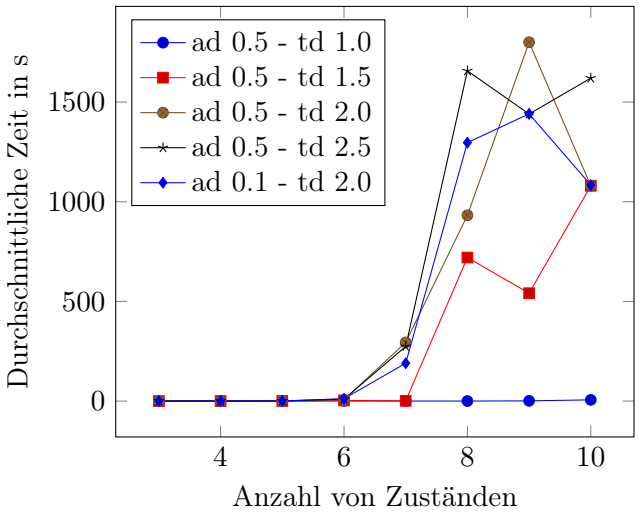


Abbildung 6.6: Verschiedene Parameter im Vergleich

Bei sehr geringer `transition density` werden meistens nicht-universelle Automaten generiert. Für diese Automaten kann die Formel oft sehr schnell gelöst werden und auch für größere Automaten von bis zu 100 Zuständen kann die zugehörige Formel oft in unter zwei Minuten gelöst werden. Bei größeren Automaten mit 200 oder mehr Zuständen scheitert der QBF-Solver QuAbs am verfügbaren Arbeitsspeicher (etwa 13Gb).

Vorverarbeitung durch RABIT

Um bessere Ergebnisse zu erreichen, können die Eingabeautomaten zunächst durch RABIT verkleinert werden (Siehe Kapitel 3.4). Viele Automaten lassen sich auf diese Weise stark verkleinern, oft werden dabei triviale Automaten mit nur einem Zustand erzeugt. Da dieser Vorverarbeitungsschritt nur wenig Zeit benötigt, die Anzahl von Zuständen und somit die Größe der Eingabe allerdings oft stark reduziert wird, können Algorithmen die auf diesen Automaten arbeiten stark davon profitieren.

Universalität und Inklusion kann mittels QBF-Solvern durch diesen Schritt für ursprünglich wesentlich größere Instanzen gelöst werden. Ist RABIT jedoch nicht in der Lage die Automaten (für Universalität) auf weniger als 8 bis 9 Zustände zu reduzieren kann die generierte Formel in den meisten Fällen nicht innerhalb von 30 Minuten gelöst werden. Ausnahme sind hier weiterhin Instanzen für die nicht-Universalität leicht gezeigt werden kann.

6.2 Formelgröße und -struktur

Bevor in Kapitel 6.3 die vorgestellten Varianten (Kapitel 4.3) mit den Ergebnissen der ursprünglichen Kodierung verglichen werden, folgt eine Diskussion über einige Eigenschaften der generierten Formeln. Zu diesem Zweck werden die Formeln für Universalität betrachtet. Die Formeln für Inklusion verhalten sich sehr ähnlich. Da für Inklusion mehr Gegenbeispiele untersucht werden müssen, sind die generierten Formeln für Inklusion allerdings größer.

Die Anzahl der logischen Operatoren der generierten Formeln wächst mit zunehmender Zustandszahl schnell an (Siehe Abbildung 6.7). Im Vergleich mit Formeln, die für den jährlichen Wettbewerb QBFEval[5] verwendet werden, sind die generierten Formeln für Automaten mit 10 bis 100 Zuständen jedoch verhältnismäßig klein.

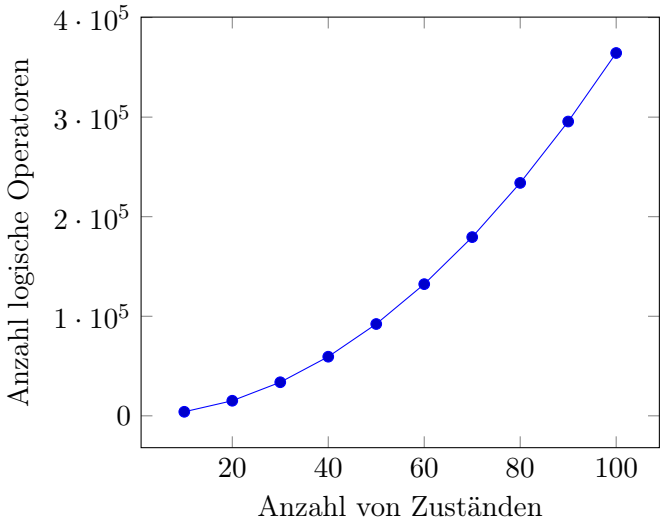


Abbildung 6.7: Größe der generierten Formeln (Universalität)

Die Parameter, mit denen der Automat generiert wurde, haben dabei nur einen geringen Einfluss auf die Größe der Formel. Ausschlaggebend ist vor allem die Größe des Automaten. (Siehe Abbildung 6.8)

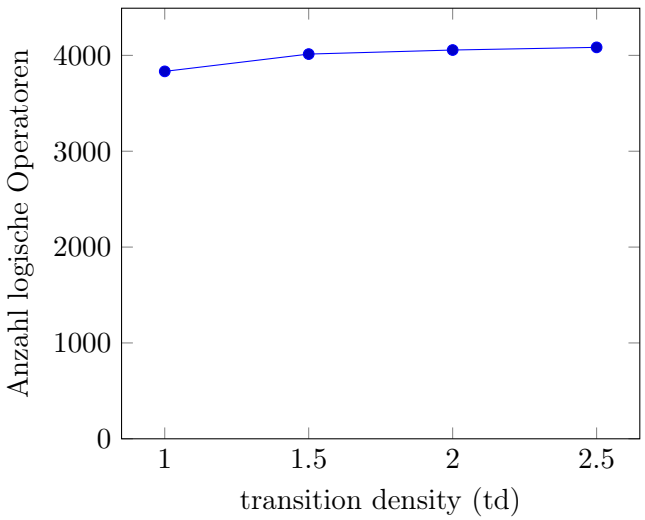


Abbildung 6.8: Größe der generierten Formeln für Automaten der Größe 10 (Universalität)

Auffällig im Vergleich mit ähnlich großen Formeln (bezüglich der Anzahl an Operatoren) der QBFEval 2017 ist, dass für Universalität (und Inklusion) über eine große Anzahl von Variablen abwechselnd quantifiziert wird. Für Universalität bei Automaten mit 6 Zuständen werden 120 Variablen quantifiziert. Die Alternierungstiefe der Quantoren, also wie oft die Art des Quantors gewechselt wird, beträgt für diese Automaten 12. Nur wenige Formeln der QBFEval 2017 verwenden ähnlich viele oder mehr Quantorenalternierungen, wie Formeln für Universalität (oder Inklusion) vergleichbarer Größe.

[5] QBFEval: `http://www.qbflib.org/index_eval.php`

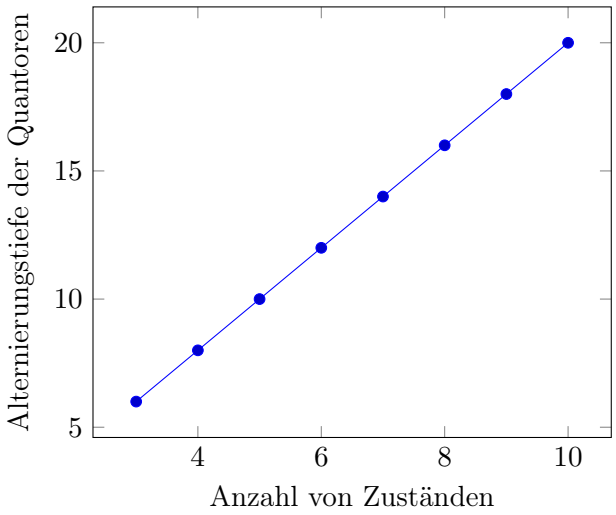


Abbildung 6.9: Wechsel zwischen All- und Existenzquantoren (Universalität)

Um die Schwierigkeit der generierten QBFs besser mit den Testdaten der QBFEval 2017 in Beziehung setzen zu können, werden als nächstes Formeln mit ähnlicher geschätzter *Baumweite* verglichen. Die Baumweite eines Graphen gibt dessen Ähnlichkeit zu einem Baum an. In einer Arbeit von Pulina und Tacchella werden QBFs als Graphen interpretiert, um deren Baumweite zu ermitteln [PT08]. Die Baumweite einer QBF kann dann verwendet werden, um deren Schwierigkeit abzuschätzen. Da das berechnen der Baumweite eines Graphen `NP`-vollständig ist, wird in der Arbeit von Pulina und Tacchella ein Werkzeug `QuTE` (Quantified Treewidth Estimator) zur Annäherung dieses Wertes vorgestellt.

Für die 90 kleinsten Formeln dieses Datensatzes[6] wurden hier diese geschätzte Baumweite berechnet. (Größere Formeln des Datensatzes verwenden wesentlich mehr Operatoren als die erzeugten Formeln für Universalität und deshalb nicht so gut vergleichbar). Zum Vergleich dienen die zehn generierten Formeln zur Universalität bei Automaten mit 4 Zuständen. Da `QuTE` nur Formeln in KNF verarbeiten kann, mussten diese Formeln in das `.qdimacs`-Format konvertiert werden. Im Vergleich mit QBF-Solvern, die das `.qcir`-Format verwenden, sind die getesteten Solver für Formeln in KNF deutlich langsamer. (Siehe Abbildung 6.5)

Für die Formeln zur Universalität für Automaten mit 4 Zuständen wurde eine durchschnittliche, geschätzte Baumweite von 51.9 (Standardabweichung von ≈ 3.7) berechnet. Der Solver DepQBF[7] für Formeln in KNF-Form benötigte im Schnitt $\approx 16.4s$ (Standardabweichung von ≈ 8.1).

[6] QBFEval 2017: `http://www.qbflib.org/eval17.zip`
[7] DepQBF: `http://lonsing.github.io/depqbf/`

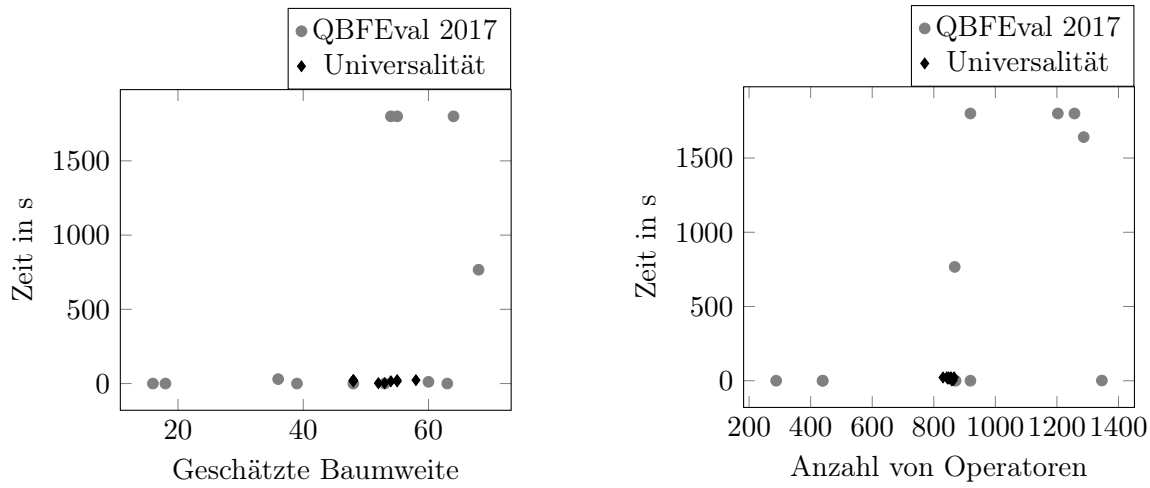


Abbildung 6.10: Zeit für ähnliche Formeln (Automatengröße 4)

Abbildung 6.10 zeigt, wie sich die generierten QBFs für Automaten mit 4 Zuständen zu ähnlichen Formeln (bezüglich Baumweite (links) und Anzahl von Operatoren (rechts)) aus dem Datensatz der QBFEval 2017 verhalten. Im Vergleich mit Formeln ähnlicher Baumweite können die generierten QBFs oft ähnlich schnell oder schneller gelöst werden. Eine ähnliche Beobachtung ergibt sich im Hinblick auf Formeln mit einer ähnlichen Anzahl logischer Operatoren.

Diese Beobachtungen liefern ein Indiz dafür, dass die generierten Formeln in Hinsicht auf andere Faktoren (als Baumweite und Größe) leichter zu lösen sind, als vergleichbare Formeln. Um die Kodierung zu verbessern, ist eine Optimierung der Größe und Baumweite der Formeln vielversprechend.

Für größere Automaten mit fünf oder mehr Zuständen wird das Timeout von 30 Minuten für jede getestete Instanz erreicht (Siehe Abbildung 6.11). Ob sich diese Beobachtung auch für größere Formeln fortsetzt, ist hier nicht direkt ersichtlich.

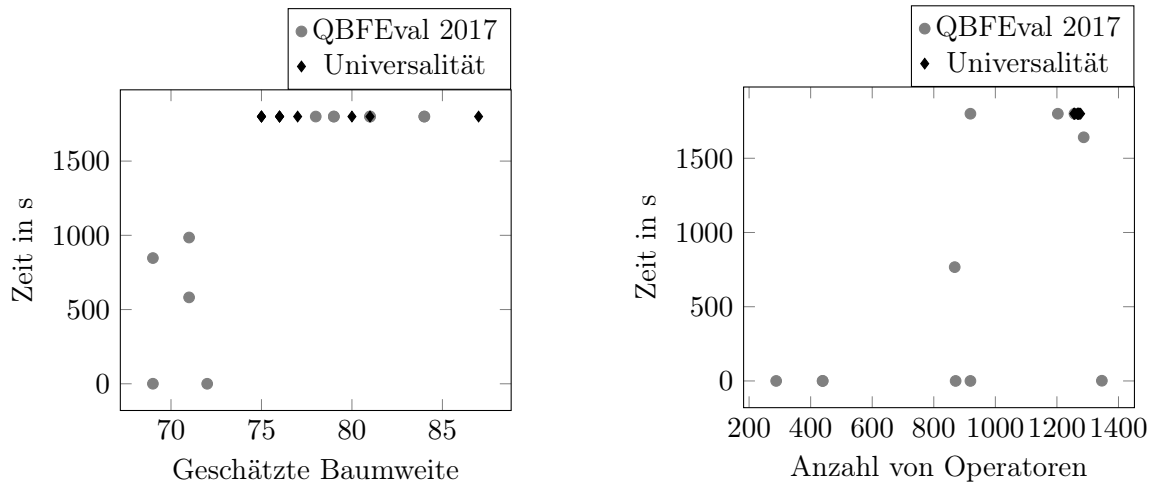


Abbildung 6.11: Zeit für ähnliche Formeln (Automatengröße 5)

Neben dem Werkzeug `QuTE` zum Schätzen der Baumweite wurde im angesprochenen Paper von Pulina und Tacchella außerdem das Programm `QuBIS` vorgestellt. `QuBIS` (Quantified Boolean formula Incomplete Solver) versucht, eine QBF entweder direkt zu lösen oder zu vereinfachen. Dabei werden oft Formeln mit geringerer *treewith* erzeugt als die Eingabeformel [PT08].

Für die generierten Formeln für Universalität verringert die Verwendung von `QuBIS` die geschätzte Baumweite nur gering (meistens um weniger als 10%). Dennoch können die verkleinerten Formeln merkbar schneller gelöst werden. Für Automaten mit 4 Zuständen verringert sich die durchschnittlich benötigte Zeit von $\approx 16.4s$ auf $\approx 10.3s$. Für größere Automaten wird weiterhin bei allen Formeln das Timeout von 30 Minuten erreicht.

6.3 Anpassungen der Formel

In Kapitel 4.3 wurden einige Varianten der Konstruktion vorgestellt. Durch diese Varianten sollte es dem Solver erleichtert werden, den Suchraum für die Belegung der Variablen zu verkleinern. Im Folgenden wird der Einfluss dieser Varianten auf die Zeit, die zum Lösen der Formeln benötigt wird, untersucht. Hier wird weiterhin nur Universalität betrachtet. Allerdings lassen sich alle Anpassungen auch, mit ähnlichen Ergebnissen, für Inklusion verwenden.

Wie bereits in Kapitel 4.3.5 beschrieben, werden für die Varianten größere Formeln generiert. (Siehe Abbildung 6.12)

Die zweite Variante benötigt dabei weniger Operatoren als die erste und dritte Variante, da ein Iterationsschritt innerhalb der $reach_i$ Teilformel eingespart werden kann.

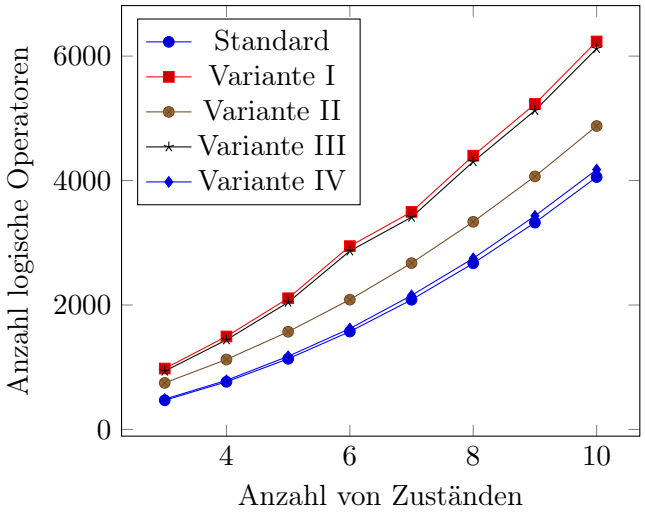


Abbildung 6.12: Größe der generierten Formeln (Universalität)

Die in Kapitel 4.3.2 geäußerte Vermutung bezüglich der benötigten Zeit zum Lösen der Formeln bestätigt sich (Siehe Abbildung 6.13). Keine der ersten drei Varianten wird effizienter gelöst als die ursprüngliche Kodierung. Da die generierten Formeln der vorgestellten Varianten größer sind, als die Formeln der einfachen Kodierung, benötigen die QBF-Solver erwartungsgemäß mehr Zeit, um diese zu lösen.

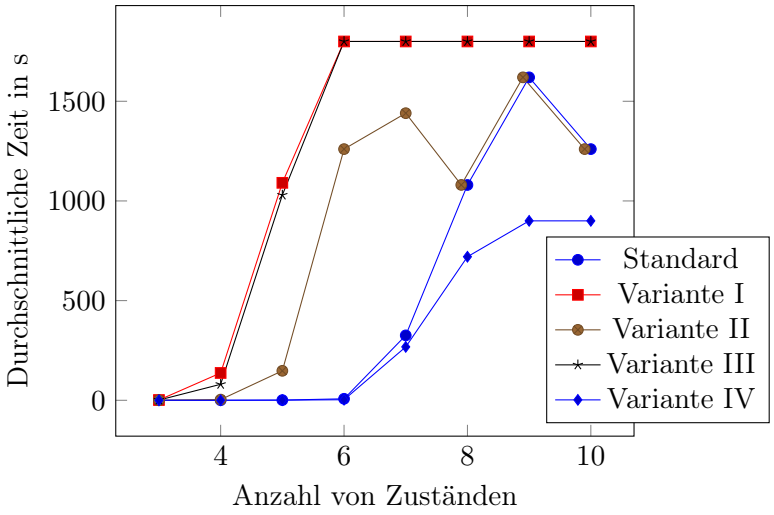


Abbildung 6.13: Benötigte Zeit zum Lösen von Universalität (QuAbs)

Variante II ist dabei merklich einfacher zu lösen, als die anderen beiden Varianten. Dies lässt sich darauf zurückführen, dass diese Variante im Vergleich zu Variante I und Variante III wesentlich kleiner ist, da ein Iterationsschritt der $reach_i$ Formel eingespart werden kann (Siehe Kapitel 4.3.2).

Die vierte Variante ist dabei die einzige Variante, die weniger Zeit benötigt als die ursprüngliche Kodierung. Diese Zeiteinsparungen treten nur bei nicht universellen Automaten auf. Im Vergleich zu bekannten Verfahren ist selbst Variante IV immer noch wesentlich langsamer. Durch die in Kapitel 4.3.4 vorgestellte vierte Variante ist es möglich, Nein-Instanzen in größeren Automaten (teilweise mit bis zu 100 Zuständen) innerhalb von 30 Minuten zu identifizieren. Existierende Implementierungen bekannter Verfahren lösen Universalität auf diesen Automaten allerdings in weniger als 5 Sekunden.

6.4 Zusammenfassung

Im Vergleich mit existierenden Implementierungen für die Entscheidungsprobleme Universalität und Inklusion benötigen aktuelle QBF-Solver wie QuAbs wesentlich mehr Zeit zum Lösen der entsprechenden Formeln. Für Universalität werden die meisten Formeln bei Automaten ab 8 Zuständen nicht mehr innerhalb von 30 Minuten gelöst. Für Inklusion wird sogar schon bei Automaten mit 5 Zuständen das Timeout erreicht. Existierende Implementierungen wie z. B. GOAL oder RABIT benötigen selbst bei wesentlich größeren Automaten nur einen Bruchteil der Zeit.

Auch einige untersuchte Umformungen der Formel liefern kein besseres Ergebnis. Lediglich für nicht-universelle Automaten konnte in Kapitel 4.3.4 eine wirksame Verbesserung der Kodierung gefunden werden. Diese ist allerdings auch im besten Fall weiterhin den etablierten Ansätzen zum Lösen dieser Entscheidungsprobleme weit unterlegen.

Die generierten Formeln scheinen für QBF-Solver sehr schwer zu lösen zu sein. Aktuelle, in QBF-Solvern implementierte, Strategien scheinen den Suchraum für die generierten Formeln bei universellen Automaten nicht genügend eingrenzen zu können.

Eine Reduzierung der Formelgröße oder der Baumweite könnte hier möglicherweise zu besseren Ergebnissen führen (siehe Kapitel 6.2). Eine automatisierte Vereinfachung der Formeln durch ein Werkzeug wie `QuBIS` ist hier allerdings nicht ausreichend.

7 Fazit und Ausblick

Universalität und Inklusion für nicht-deterministische endliche Automaten sind zwei PSpace-vollständige Entscheidungsprobleme der Informatik. Der klassische Ansatz benötigt den Potenzautomaten, um diese Probleme zu entscheiden. Da dessen Konstruktion sehr aufwendig ist, versuchen moderne Verfahren [DDHR06, BP13] diesen Schritt weitestgehend zu vermeiden. In vielen Fällen kann es so vermieden werden, den kompletten Potenzautomaten zu konstruieren.

In dieser Arbeit wurde ein weiterer alternativer Ansatz vorgestellt. Aufbauend auf der Potenzmengenkonstruktion können die Entscheidungsprobleme Universalität und Inklusion für nicht-deterministische endliche Automaten in eine quantifizierte boolesche Formel kodiert werden. Für das Erfüllbarkeitsproblem von QBF gibt es viele verschiedene Verfahren [LE17, Ten16], welche in der Lage sind, auch sehr große Formeln in kurzer Zeit zu lösen (siehe Kapitel 2). Da dieses Problem ebenfalls PSpace-vollständig ist, benötigen diese Verfahren im schlimmsten Fall auch exponentiell viel Zeit.

Zum effizienten Entscheiden von Universalität und Inklusion ist eine Kodierung in QBF nicht geeignet. Viele der existierenden Algorithmen zum Lösen dieser Probleme verwenden exponentiellen Platz und lassen sich demnach nicht als QBF kodieren. Die vorgestellte Kodierung basiert auf der Potenzmengenkonstruktion und benötigt wesentlich mehr Zeit als existierende Implementierungen (anderer Algorithmen), wie z. B. GOAL [TCT+08] oder RABIT [MC13] (siehe Kapitel 6.1).

Es wurden einige Variationen der Kodierung untersucht. Durch eine kleine Modifikation konnte dabei die Geschwindigkeit für das Suchen von Gegenbeispielen erhöht werden. Keine dieser Modifikationen der Konstruktion konnte bei universellen Automaten (bzw. enthaltenden Automaten) bessere Ergebnisse erzielen.

Der Großteil der verwendeten logischen Gatter und Quantoren wird erzeugt, um einen Pfad innerhalb des Automaten zu suchen. Alle Varianten der Konstruktion verwenden die gleiche (bzw. sehr ähnliche) Teilformel $reach_i$, um diesen Pfad zu finden. Da die Größe dieser Teilformel quadratisch mit steigender Zustandszahl zunimmt, besteht dort der größte Optimierungsbedarf. Gleichzeitig werden für den vorgestellten Ansatz zu Universalität und Inklusion Gegenbeispiele exponentieller Länge benötigt. Dementsprechend gibt es nur wenig Spielraum, um die Konstruktion in dieser Hinsicht zu verbessern.

Neben der Optimierung der Größe der Formeln ist außerdem eine Reduzierung der Baumweite erfolgversprechend (Siehe Kapitel 6.2), um die benötigte Zeit zu reduzieren. Ein zusätzlicher Ansatz für eine weiterführende Forschung wäre eine direkte Kodierung in konjunktive Normalform (KNF). Dadurch könnten KNF-Solver die Formeln möglicherweise wesentlich schneller lösen, als Formeln die automatisiert in KNF konvertiert wurden.

Literaturverzeichnis

[BHPS61] Bar-Hillel, Yehoshua ; Perles, M. ; Shamir, E.: On Formal Properties of Simple Phrase Structure Grammars. In: *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* 14 (1961), S. 143–172. – Reprinted in Y. Bar-Hillel. (1964). *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley 1964, 116–150

[BP13] Bonchi, Filippo ; Pous, Damien: Checking NFA equivalence with bisimulations up to congruence. In: *ACM SIGPLAN Notices* Bd. 48 ACM, 2013, S. 457–468

[DDHR06] De Wulf, Martin ; Doyen, Laurent ; Henzinger, Thomas A. ; Raskin, J-F: Antichains: A new algorithm for checking universality of finite automata. In: *International Conference on Computer Aided Verification (CAV)* Springer, 2006, S. 17–30

[HK71] Hopcroft, John E. ; Karp, Richard M.: A linear algorithm for testing equivalence of finite automata / Cornell University. 1971. – Forschungsbericht

[HK09] Holzer, Markus ; Kutrib, Martin: Descriptional and computational complexity of finite automata. In: *International Conference on Language and Automata Theory and Applications* Springer, 2009, S. 23–42

[Hop71] Hopcroft, John: An n log n algorithm for minimizing states in a finite automaton. In: *Theory of machines and computations*. Elsevier, 1971, S. 189–196

[JR93] Jiang, Tao ; Ravikumar, Bala: Minimal NFA problems are hard. In: *SIAM Journal on Computing* 22 (1993), Nr. 6, S. 1117–1141

[KSGC10] Klieber, William ; Sapra, Samir ; Gao, Sicun ; Clarke, Edmund: A non-prenex, non-clausal QBF solver with game-state learning. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)* Springer, 2010, S. 128–142

[LE17] Lonsing, Florian ; Egly, Uwe: DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In: *International Conference on Automated Deduction* Springer, 2017, S. 371–384

[MC13] Mayr, Richard ; Clemente, Lorenzo: Advanced automata minimization. In: *ACM SIGPLAN Notices* Bd. 48 ACM, 2013, S. 63–74

[MF71] Meyer, A. R. ; Fischer, M. J.: Economy of description by automata, grammars, and formal systems. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1971. – ISSN 0272–4847, S. 188–191

[PT08] Pulina, Luca ; Tacchella, Armando: Treewidth: A useful marker of empirical hardness in quantified boolean logic encodings. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning* Springer, 2008, S. 528–542

[RS59] Rabin, Michael O. ; Scott, Dana: Finite automata and their decision problems. In: *IBM journal of research and development* 3 (1959), Nr. 2, S. 114–125

[Sav70] Savitch, Walter J.: Relationships between nondeterministic and deterministic tape complexities. In: *Journal of computer and system sciences* 4 (1970), Nr. 2, S. 177–192

[SM73] Stockmeyer, Larry J. ; Meyer, Albert R.: Word problems requiring exponential time (preliminary report). In: *Proceedings of the fifth annual ACM symposium on Theory of computing (STOC)* ACM, 1973, S. 1–9

[TCT+08] Tsay, Yih-Kuen ; Chen, Yu-Fang ; Tsai, Ming-Hsien ; Chan, Wen-Chin ; Luo, Chi-Jian: GOAL extended: Towards a research tool for omega automata and temporal logic. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* Springer, 2008, S. 346–350

[Ten16] Tentrup, Leander: Solving QBF by abstraction. In: *arXiv preprint arXiv:1604.06752* (2016)

[TV05] Tabakov, Deian ; Vardi, Moshe Y.: Experimental evaluation of classical automata constructions. In: *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* Springer, 2005, S. 396–411

[WTJK17] Wang, Hung-En ; Tu, Kuan-Hua ; Jiang, Jie-Hong R. ; Kushik, Natalia: Homing Sequence Derivation with Quantified Boolean Satisfiability. In: *IFIP International Conference on Testing Software and Systems* Springer, 2017, S. 230–242

Akronyme

DEA deterministischer endlicher Automat.

DNF disjunktive Normalform.

KNF konjunktive Normalform.

NEA nicht-deterministischer endlicher Automat.

QBF quantifizierte boolesche Formel.