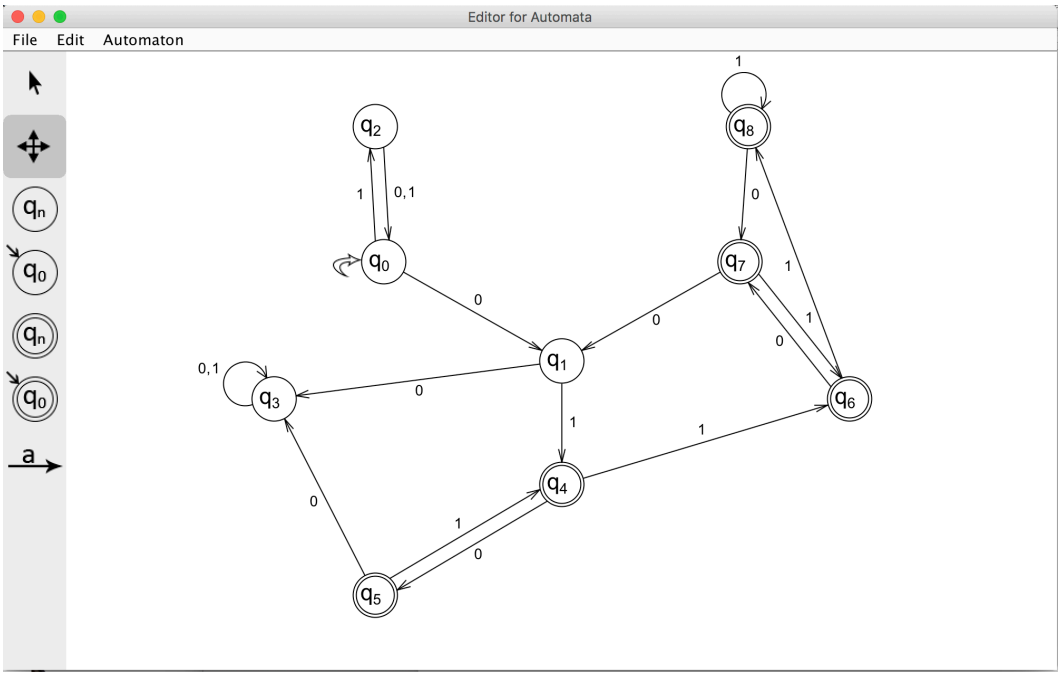
Universität Bern
Institut für Informatik
FS 2017
Prof. Dr. Thomas Studer

Bachelorarbeit

Editor for Automata

Softwaredokumentation



Raphael Bucher
07-559-529
Brünnenstrasse 62
3018 Bern
raphael.bucher1@students.unibe.ch
20. April 2017

Inhaltsverzeichnis

Einleitung

Im Verlaufe der Vorlesung „Automaten und formale Sprachen“ des Studiengangs „Bachelor of Science in Computer Science“ der Universität Bern kam mir der Gedanke, dass man für dieses Thema hervorragend einen kleinen Editor programmieren könnte. Mit diesem sollte es möglich sein, Automaten zu konstruieren und deren Layout viel komfortabler zu verändern, als es auf Papier möglich ist. Ferner könnte man Automaten speichern und laden, sie als Bild exportieren, sie in andere Typen umwandeln (nichtdeterministischer endlicher Automat, deterministischer endlicher Automat, minimaler deterministischer endlicher Automat), eine äquivalente Grammatik vom Typ-3 anzeigen lassen, testen ob der Automat ein bestimmtes Wort akzeptiert etc.
Ich entschied, einen solchen Editor in Java zu programmieren im Rahmen meiner Bachelorarbeit „Editor for Automata“. Ein schönes GUI und Benutzerfreundlichkeit im Allgemeinen waren zentrale Anliegen für mich.
Bezüglich den Automaten habe ich die grundlegenden Typen „Deterministischer endlicher Automat“, „Minimaler Deterministischer endlicher Automat“, „Nichtdeterministischer endlicher Automat“ und „Epsilon-Automat“ abgedeckt. Der Editor deckt die gängigsten Algorithmen zu diesen Typen ab. Das Programm enthält diverse Tooltips, welche dem Benutzer die Handhabung auf einfache und schnelle Art erklären.
Die Arbeit lehnt sich an das Skript der oben genannten Vorlesung an. Das Skript ist online nicht verfügbar, grundlegende Informationen zur Vorlesung finden sich unter:
http://www.ltg.unibe.ch/lectures/FS17/AFS

Der Programmcode und eine ausführbare Version des Editors sind auf meinem GitHub-Account:
https://github.com/RaphaelBucher/EditorForAutomata

Funktionsumfang

1. Automatenkonstruktion

a. Automatenzustand hinzufügen

Ein Automatenzustand kann hinzugefügt werden, in dem der Benutzer einen der vier Buttons „Add State Tool“, „Add Start State Tool“, „Add End State Tool“ oder „Add Start & End State Tool“ der Werkzeugleiste am linken Rand auswählt. Dabei wird der Mauscursor gewechselt, wenn der Benutzer die Maus wieder in die weisse „Zeichnungsfläche“ des Automaten bewegt. Durch einen weiteren Linksklick wird ein Zustand des aktuell ausgewählten Typs hinzugefügt.
Aufgrund der Definition eines Automaten ist nur ein Startzustand erlaubt. Falls der Benutzer versucht, einen zweiten Startzustand hinzuzufügen, wird diese Aktion nicht ausgeführt, und stattdessen ein Hinweis angezeigt, dass dies untersagt ist. Für die restlichen Typen gelten keine Restriktionen.
Die Positionierung der Zustände ist dem Benutzer überlassen, übereinander gestapelte Zustände können später wieder auseinandergeschoben werden. Die grafische Darstellung ist derjenigen des Skripts der Vorlesung nachempfunden. Bei der Indexierung ist die 0 für den Startzustand reserviert.

b. Übergang hinzufügen

Um dem Automaten einen Übergang zwischen zwei Zuständen hinzuzufügen, muss der Benutzer das „Add Transition Tool“ der Werkzeugleiste auswählen. Die eingeblendeten Hinweise am oberen Rand der Zeichnungsfläche sollten den Benutzer mühelos durch den mehrteiligen Vorgang leiten. Als nächstes muss der Benutzer einen Zustand auswählen, von dem aus der Übergang starten soll. Hat er einen Zustand ausgewählt, wird dieser zwecks optischer Übersicht markiert.

Nun muss der Benutzer einen Zustand auswählen, zu dem der Übergang führen soll. Dies kann auch der bereits ausgewählte Zustand sein. Falls der Benutzer in diesem Schritt keinen Zustand ausgewählt hat, landet er wieder beim ersten Schritt (die Markierung des ersten Zustandes wird entfernt). Dies ermöglicht dem Benutzer, falsche Eingaben schnell zu korrigieren.

Im letzten Schritt kann der Benutzer eines oder mehrere Zeichen hinzufügen. Erlaubt sind die Zeichen a bis z und 0 bis 9. Das leere Wort ε wird mittels der Leertaste hinzugefügt. Jedes Zeichen ist nur einmal erlaubt. Ein Mausklick an eine beliebige Position beendet den Vorgang und führt den Benutzer wieder zu Schritt eins. Nur falls mindestens ein gültiges Zeichen eingegeben wurde, wird der Übergang dem Automaten hinzugefügt. Der Automat befindet sich zu jeder Zeit in einem per Definition des Skripts gültigen Zustand. Dies reduziert die Anzahl der Tests auf gültige Automaten beim Speichern und Laden eines Automaten, beim Starten der Transformations-Algorithmen oder dem Test eines Wortes auf Akzeptanz.

c. Einem bereits erstellten Übergang ein weiteres Zeichen hinzufügen

Falls der Benutzer einem bereits erstellten Übergang später weitere Zeichen hinzufügen will, kann er dies auf die gleiche Art wie in Punkt b beschrieben bewerkstelligen. Es gelten die gleichen Richtlinien (kein Zeichen doppelt etc.).

d. Löschen von Zuständen, Übergängen und Zeichen

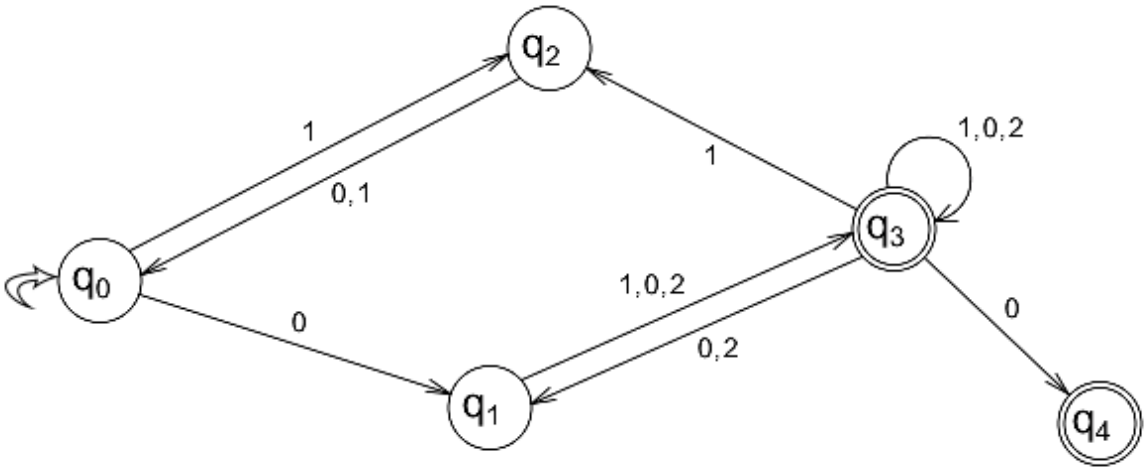
Um einen Zustand zu löschen, muss der Benutzer das „Selection Tool“ der Werkzeugleiste auswählen. Anschliessend kann ein Zustand, ein Übergang oder ein Zeichen eines Übergangs mittels der Maus ausgewählt werden. Es können nicht mehrere Sachen gleichzeitig ausgewählt werden. Falls sich Zustände, Übergänge und Zeichen visuell überlappen, haben Zeichen die höchste Priorität, gefolgt von

Übergängen und Zuständen. Bei der Auswahl von Übergängen gibt es 6 Pixel Toleranz, der Benutzer muss die schmale Linie des Übergangs nicht exakt treffen mit der Maus. Falls ein Übergang ausgewählt wird, werden auch alle seine Zeichen ausgewählt bzw. markiert.

Falls der Benutzer etwas ausgewählt hat, kann er anschliessend gemäss Hinweis des Programms mittels „Backspace“ das ausgewählte Objekt löschen. Beim Löschen eines Übergangs werden auch alle seine Zeichen gelöscht. Falls ein Zustand gelöscht wird, werden alle Übergänge von und zu dem Zustand gelöscht. Die Indexierung der Zustände bleibt dabei erhalten. Lücken in der Indexierung können später mit neuen Zuständen wieder aufgefüllt werden.

Auf die grafischen Auswirkungen dieser Aktionen wird später eingegangen.

2. Visualisierung



a. Darstellung von Zuständen, Übergängen und Zeichen

Die visuelle Darstellung von Zuständen, Übergängen und Zeichen ist derjenigen des Skripts nachempfunden. Endzustände haben eine

doppelte Umkreisung, die restlichen nur einen Kreis. Die Darstellung des „Pfeils“ des Startzustandes weicht etwas von derjenigen des Skripts ab, sollte aber dennoch leicht erkennbar sein.

Übergänge zwischen verschiedenen Zuständen werden ausschliesslich durch gerade Linien mit einem Pfeil repräsentiert, kurvige Linien von Beispielen aus dem Skript sind nicht möglich. Der Hauptgrund für diese Designentscheidung liegt in der Abfrage für das Auswählen eines Übergangs durch einen Mausklick. Für die gerade Linie eines Überganges zwischen zwei verschiedenen Zuständen reicht die Formel des Abstands eines Punktes (Mauszeiger) zu einer Geraden (gerade Linie des Übergangs), kombiniert mit dem Abstand des Mauszeigers zur Mitte des Übergangs da keine unendliche Gerade vorliegt. Beim zweiten Kriterium muss der Mauszeiger innerhalb des Kreises liegen, welche die gerade Linie des Übergangs als Durchmesser hat. Falls beide Kriterien zutreffen, befindet sich der Mauszeiger über der geraden Linie des Übergangs. Beim ersten Kriterium habe ich eine Toleranz von sechs Pixel eingebaut, da ein Anwählen sonst zu fummelig ausfallen würde.

Bei freier Linienführung würde sich die Kollisionsabfrage und die Positionierung der Zeichen des Übergangs als schwierig erweisen.

Verschiebt man zwei Zustände nahe genug zueinander, wird der Übergang mit all seinen Zeichen temporär ausgeblendet zwecks Darstellung. Ausserdem ist er nicht mehr anwählbar.

Übergänge zwischen zwei gleichen Zuständen werden als Teilkreise mit 270° dargestellt. Dies ermöglicht eine einfache Ausrichtung, und die Positionierung der Zeichen und die Kollisionsabfrage gestaltet sich ebenfalls leicht. Auch hier gibt es wieder die sechs Pixel Toleranz bei der Kollisionsabfrage, damit der Benutzer den Teilkreis nicht exakt mit der Maus treffen muss.

Die Zeichen eines Überganges werden immer in der Mitte der Linie dargestellt, leicht erkennbar durch Verschieben eines Zustandes mit einem Übergang. Je nach Winkel werden die Zeichen entweder nach links oder nach rechts ausgeschrieben. Für die Kollisionsabfrage und die folgende Markierung von Zeichen wurde Javas FontMetrics-Klasse verwendet, welche Informationen wie Breite und Höhe von einzelnen

Buchstaben einer Schrift liefert. Damit konnten unterschiedlich breite Buchstaben (Zeichen) bei Auswahl passend hellblau einschattiert werden.

Das GUI wurde mit Javas Swing-framework realisiert. Weitere Frameworks wurden nicht verwendet, die Berechnungen für die Darstellung von Zuständen, Übergängen und Zeichen wurde selber geschrieben. Ein etwaiger Mehraufwand habe ich Zugunsten einer vollen Kontrolle meinerseits in Kauf genommen. Mit dieser Entscheidung war ich jedoch nie unglücklich, da dieser Teil wenig Probleme bereitete.

b. Auswählen von Zuständen, Übergängen und Zeichen

Die oben genannten Objekte können, wie bereits beschrieben, ausgewählt werden und haben eine blaue Linienführung bzw. Schattierung zur Folge. Eine Markierung kann durch einen Klick auf etwas Anderes abgewählt werden. Je nach markiertem Objekt erscheint ein entsprechender Hinweis für die Löschung des Objekts am oberen Rand der Zeichenfläche.

c. Verschieben von Zuständen

Um einen Zustand zu verschieben, muss der Benutzer das „Move Tool“ auswählen und anschliessen einen Zustand per „drag and drop“ verschieben.

Ein Verschieben von einem Zustand löst eine Aktualisierung aller grafischen Berechnung von diesem Zustand, seinen Übergängen und allen „benachbarten“ Zuständen aus (Zustände, welche mit dem verschobenen Zustand durch einen Übergang verbunden sind). Die Wirkungsweise kann leicht eingesehen werden, in dem man den Automaten 02_demo.xml öffnet und beispielsweise den Zustand q_1 verschiebt. Man beachte, dass sich die runden Übergänge zwischen gleichen Zuständen jeweils die grösstmögliche Lücke für ihre Positionierung suchen.

3. File-Menu

a. Neuer Automat

Ein Klick auf den Menü Eintrag „New“ erstellt erwartungsgemäss einen neuen, leeren Automaten ohne jegliche Zustände, Übergänge und Zeichen.

b. Automat speichern

Der Benutzer kann seine Automaten speichern mittels des Menü Eintrags „Save“. Beim folgenden Dialog wird ihm ein vorgefertigter Ordner vorgeschlagen, in dem bereits mehrere Demo-Automaten liegen. Der Benutzer kann jedoch einen beliebigen Speicherort wählen. Beim Format habe ich die XML-Variante gewählt, da Java viele nützliche Klassen hierfür bereitstellt und sich XMLs Baumstruktur anbietet. Beispiele können im Ordner „SavedAutomata“ eingesehen werden. Zur Speicherung nötig sind lediglich die Zustände mit ihren Indices, ihrem Typ (Startzustand, Endzustand etc.) und ihren Koordinaten, und die Übergänge mit ihren zwei verbundenen Zuständen und einer Liste von den Zeichen, die sie beherbergen.

c. Automat öffnen

Mittels dem Menü Eintrag „Open“ kann ein gespeicherter Automat geöffnet werden. Es werden verschiedene Überprüfungen hinsichtlich einer korrekten Datei vorgenommen. Die visuelle Repräsentation ist exakt dieselbe wie beim gespeicherten Automaten (nach dem Laden der Datei wird ein komplettes grafisches update aller Elemente gemacht).

d. Bild-Export

Es ist möglich, seinen erstellten Automaten als PNG-Bild zu exportieren. Allfällige aktive Hinweise am oberen Rand der Zeichnungsfläche oder sonstige Meldungen werden für den Image-Export ausgeblendet. Mittels grafischer Optionen wie Kantenglättung konnte die Bildqualität deutlich verbessert werden. Der Inhalt des Graphics2D-Objektes des JPanels (Zeichnungsfläche) wurde direkt in ein BufferedImage-Objekt gegossen für die anschliessende Speicherung.

4. Edit-Menu

a. Rückgängig / Wiederholen

Die meisten Aktionen des Benutzers sind umkehr- und wiederholbar. Falls eine Aktion umkehrbar ist, dann erscheint nach ihrer Ausführung im „Undo“-Menü ein passender Eintrag.
Hierfür wurden die gewohnten, betriebssystemspezifischen Hotkeys „cmd+z“ etc. konfiguriert. Umkehrbar sind folgende Aktionen:
-Zustand hinzufügen / löschen
-Übergang hinzufügen / löschen
-Zeichen hinzufügen / löschen
-Zustand verschieben
-Layout-Algorithmus ausführen
-Unerreichbare Zustände entfernen
-Automaten in einen NEA umwandeln
-Automaten in einen DEA umwandeln
-Automaten in einen minimalen DEA umwandeln

Für kleinere Aktionen, wie einen Zustand hinzufügen, wurden nur die hierfür notwendigen Änderungen am Automat in die Historie aufgenommen. Aktionen, welche den ganzen Automaten umkrempeln (z.B. die Transformations-Algorithmen) jedoch lassen eine ganze Kopie

von sich registrieren, mittels derer die Aktion rückgängig gemacht werden kann.
Die „undo-redo“ Funktionalität befindet sich im controlFlow-package. Kern der Sache ist die abstrakte Klasse UserAction, welche ein Objekt der Klasse LinkedList mit den registrierten Aktionen des Benutzers enthält. Mittels undo / redo wird dann diese Liste in die entsprechende Richtung traversiert. Falls z.B. der Benutzer 10 Aktionen durchgeführt hat, fünf davon wieder rückgängig machte und nun eine neue Aktion ausführt, dann werden die letzten fünf gespeicherten Aktionen aus der Liste entfernt, und anschliessend die zuletzt gemachte Aktion drangehängt. Ebenso wird der Index auf das gerade traversierte Element aktualisiert.
Falls die Liste voll ist, wird das älteste Element entfernt. Falls ein neuer Automat geladen wird, wird die Liste gelöscht und eine neue Historie begonnen.

Die Klasse UserAction enthält auch die zwei abstrakten Methoden undo und redo, welche von erbenden Klassen dann implementiert werden nach ihrem Gusto. Die LinkedList besteht also aus Objekten der UserAction-Klasse, wobei die tatsächlichen Instanzen z.B. die erbenden Klassen AddedState, RemovedTransition etc. sind und dann selber definieren, was genau passiert bei undo / redo.

Für diese Funktionalität habe ich die Objektorientierung herangezogen. Andere Teile des Programms, wie z.B. die ganzen Transformations-Algorithmen („Transform to NFA“ etc.) sind jedoch statisch. Hier habe ich keinen Mehrwert gesehen, noch ein einzelnes Objekt von diesen Klassen zu erzeugen, nur um nachher darauf die besagten Methoden aufzurufen.

Ob eine Aktion registriert werden soll für die undo-redo Funktionalität, hängt von ihrem Kontext ab. Fügt z.B. der Benutzer manuell einen Zustand hinzu, so wird sie registriert. Wird diese Methode jedoch im Rahmen von „load Automat“ bei der Konstruktion des geladenen

Automaten aufgerufen, so soll sie nicht registriert werden, da diese Aktion nicht umkehrbar sein soll. (sonst wäre in der Historie zigmal „AddedState“, „AddedTransition“ etc. drin). Ich habe das Problem mittels einem zusätzlichen Parameter bei diesen Methoden gelöst, es funktioniert, aber richtig glücklich bin ich damit nicht, da es den Quellcode ein wenig verunreinigt hat. Andererseits hatte ich keine alternative Idee zur Lösung dieses Problems.

5. Automaton-Menu

a. Information zum Automaten

Der Benutzer kann sich die Definition des Automaten gemäss Skript S. 10 einsehen. Hierbei wird zuerst der Typ des Automaten bestimmt. Meine Software deckt die endlichen Automaten gemäss Skript Kapitel 2 ab (ohne Mealy- und Moore-Maschinen), somit ist jeder konstruierbare Automat automatisch ein ε-Automat gemäss Skript Definition 2.2.9. Dementsprechend habe ich auf einen Eintrag hierzu verzichtet.
Die relevanten Typbestimmungen beschränken sich auf DEA, NEA und minimaler DEA. Während die Bestimmungen zum DEA und NEA schnell gehen, muss man zur Bestimmung, ob der Automat ein minimaler DEA ist, den Transformations-Algorithmus „Transform to minimal DFA (DEA)“ anwerfen, um anschliessend zu sehen, ob es äquivalente Zustände gab, die man zusammen vermengen könnte. Dem Algorithmus wird hierbei eine Kopie des Automaten zugespielt, um unerwünschte Manipulationen am aktuell dargestellten Automaten zu vermeiden.
Anschliessend folgt die angewandte Definition von S. 10 aus dem Skript. Man beachte, dass die Überführungsfunktion im Falle eines DEAs in die Zustandsmenge Q abbildet, während sie sonst in die Potenzmenge abbildet von Q abbilden würde.

b. Typ-3 Grammatik des Automaten

Es ist dem Benutzer möglich, die Typ-3 Grammatik anzeigen zu lassen welche die akzeptierte Sprache des Automaten generiert, sprich L(G) = A(E). Es wird eine rechtslineare Grammatik erzeugt (eine linkslineare Grammatik wäre nach Satz 7.4 des Skripts auch möglich).
Das Konstruktionsverfahren läuft nach dem Beweis von Satz 7.9 im Skript, die nötigen Ableitungen erhält man mittels folgenden Regeln:

P:

- $A \rightarrow aB$, für alle $A, B \in Q$ und $a \in \Sigma$ mit $\delta(A, a) = B$
- $A \rightarrow a$, für alle $A \in Q$, $B \in F$ und $a \in \Sigma$ mit $\delta(A, a) = B$
- $S \rightarrow \varepsilon$, falls $q_0 \in F$

Dies ergibt z.B. für den Demo-Automaten 03_demo.xml die Ableitungen:

$S \rightarrow aA \rightarrow aaB$
$S \rightarrow aA \rightarrow aa$
$S \rightarrow aA \rightarrow abB$
$S \rightarrow aA \rightarrow ab$

Und somit:

$L(G) = \{aa, ab\} = A(E)$

Analog zum Skript werden die nichtterminalen Zeichen gemäss Konvention gross und die terminalen Zeichen klein geschrieben. Wie in den Beispielen aus der Vorlesung findet eine Umbenennung der Zustände statt, ich fand folgende Systematik angebracht. (Man beachte, dass S als Startsymbol schon vergeben ist).

$q_0 = S$, $q_1 = A$, $q_2 = B$,...,$q_{18} = R$, $q_{19} = T$, $q_{20} = U$,...,$q_{25} = Z$, $q_{26} = A_1$, $q_{27} = A_2$,...

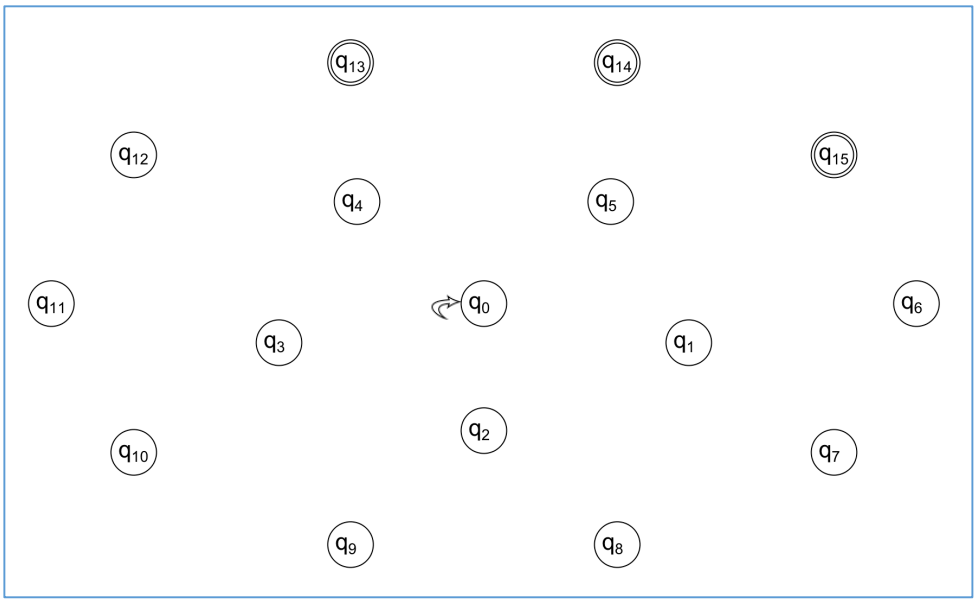
c. Layout-Algorithmus

Da bei den Transformations-Algorithmen häufig komplett neue Automaten entstehen, musste ein Layout-Algorithmus her welcher die Zustände auf akzeptable Art und Weise auf der Zeichnungsfläche anordnet. (Gerade bei der Transformation NEA nach DEA können sehr viele neue Zustände und Übergänge entstehen).

Leider hat ein Automat nicht zwingend eine Baumstruktur, denn für Bäume gäbe es viele Visualisierungstechniken, die mit vertretbarem Aufwand umsetzbar wären. Weiter zu beachten war meine Design-Entscheidung, Übergänge zwischen zwei verschiedenen Zuständen ausschliesslich als gerade Linien mit Pfeil darzustellen.

Herausgekommen ist ein Algorithmus nach eigener Idee, der in den meisten Fällen meines Erachtens gute Ergebnisse liefert, aber je nach Anzahl Zuständen und Übergängen dann doch irgendwann an seine Grenzen stösst. Nicht hinsichtlich der Performance, sondern hinsichtlich des optischen Ergebnisses. Andererseits ist dies ab einer gewissen Menge an optischen Elementen ohnehin der Fall.

Als erster Schritt werden alle Zustände sortiert nach der Anzahl ihrer Übergänge. Anschliessend wird der Zustand mit den meisten Übergängen in der Mitte angeordnet. Die restlichen werden der Reihe nach gemäss Sortierung kreisförmig um den mittleren Zustand platziert, sodass die Gesamtdistanz all ihrer Übergänge möglichst klein ist. Ausserdem findet eine Skalierung zu Ellipsen statt falls die Zeichnungsfläche nicht quadratisch ist. Der „Start-Winkel“ der Kreispositionen ist leicht versetzt, um mehrere Zustände in einer Linie zu vermeiden, die im Falle von Übergängen dann Überlappungen zur Folge hätten. Folgender Automat ohne Übergänge verdeutlicht die Anordnung:



Je nach Fenstergrösse und Seitenverhältnis liefert der Algorithmus andere Ergebnisse, beispielhaft zu sehen beim Automaten 02_demo.xml.

d. Unerreichbare Zustände entfernen

Zustände, welche vom Startzustand aus nicht erreichbar sind, können im Automaton-Menü unter dem Menü Eintrag „Remove unreachable States“ entfernt werden. Die Funktionsweise kann leicht anhand des Demo-Automaten 04_demo.xml ausprobiert werden.

Der Algorithmus basiert auf einer simplen Rekursion, in der alle vom Startzustand aus erreichbaren Zustände rekursiv markiert werden. Die Methode ruft sich nur selber auf dem Zielzustand auf, falls dieser noch nicht markiert ist. Anschliessend werden die nicht markierten Zustände entfernt.

Folgender Quellcode zeigt die Funktionsweise:

```
/** Recursive method that calls itself for all unmarked neighbor-states of that passed state
that can be reached by one transition. */
private static void markReachableNeighborStates(Automat automat, State state) {
  state.setMarked(true);

  // Get all reachable Neighbor states
  ArrayList<Transition> outgoingTransitions = Transition.getTransitionsByStartState(state,
      automat.getTransitions());

  for (int i = 0; i < outgoingTransitions.size(); i++) {
    if (!outgoingTransitions.get(i).getTransitionEnd().isMarked()) {
      markReachableNeighborStates(automat, outgoingTransitions.get(i).getTransitionEnd());
    }
  }
}
```

e. Automat in einen NEA umwandeln

Ein beliebiger, mit meiner Software konstruierbarer Automat, kann in einen NEA umgewandelt werden, welcher die gleiche Sprache akzeptiert. Falls bereits ein NEA vorliegt führt der Algorithmus keine Aktionen durch, nur wenn es Epsilon-Übergänge gibt werden diese gemäss dem Verfahren aus dem Beweis von Satz 2.2.12 eliminiert.
Der Algorithmus wurde in der Methode transformToNEA der Klasse Transformation aus dem gleichnamigen package untergebracht.

Da dieser Vorgang mittels „undo“ rückgängig gemacht werden kann, wird auf einer Kopie des aktuellen Automaten des Editors gearbeitet. Wenn der Algorithmus terminiert hat, werden die beiden Referenzen für die undo-redo Historie registriert.
Der Algorithmus arbeitet nach der Vorlage aus dem Skript, gut einsehbar anhand der Abbildung 2.2.6. Nach dem Übertragen der Menge der Zustände, dem Startzustand und den Endzuständen folgt Schritt 2 der Abbildung 2.2.6, implementiert in der Methode toNEANewEndStates der Klasse Transformation. Gemäss der Definition des Skripts werden hier alle Zustände, welche durch eine reine „Epsilon-Kette“ einen Endzustand

erreichen können, ebenfalls selber zu Endzuständen. Als Epsilon-Kette bezeichne ich eine Aneinanderreihung von Übergängen, bei welcher jeder Übergang das leere Wort, dargestellt als ε, enthält. Ob die Übergänge noch weitere Zeichen haben ist nicht von Belang.
Dieser Teil des Algorithmus iteriert über alle Endzustände, und ruft auf ihnen das rekursive Markierungsverfahren auf, halt lediglich in entgegengesetzte Richtung der Übergänge und mit der Bedingung, dass der Übergang das leere Wort ε enthält (Vgl. Punkt d, „unerreichbare Zustände entfernen“). Anschliessend werden die damit markierten Zustände in Endzustände umgewandelt: Startzustände zu Start-End Zuständen, normale Zustände zu normalen Endzuständen.

Bei Schritt 3 aus Abbildung 2.2.6 habe ich mich dazu entschieden, die benötigten Übergänge neu zu konstruieren anstelle einer Umwandlung der alten Übergänge. Dieser Schritt iteriert über alle Zustände des Automaten, erstellt für jeden Zustand eine Liste mit allen Übergängen welche in diesen Zustand münden, löscht Übergänge aus dieser Liste welche als einziges Zeichen das leere Wort ε haben, und markiert dann wieder rekursiv alle Zustände, welche rückwärts durch reine Epsilon-Ketten erreichbar sind. Von diesen markierten Zuständen aus wird dann je einen Übergang mit entsprechenden Zeichen zu dem anfänglich iterierten Zustand gebaut.
Epsilon-Zeichen werden temporär rausgefiltert, löschen kann ich sie noch nicht da sie für die weiteren Iterationen gebraucht werden. Damit ist Schritt 3 der Abbildung 2.2.6 abgeschlossen und alle Epsilon-Übergänge wurden umgewandelt.

Der optionale Schritt 4 aus Abbildung 2.2.6 wurde nicht implementiert, da der Benutzer auf Wunsch selber die unerreichbaren Zustände entfernen lassen kann. Häufig ist die Umwandlung zu einem NEA einfacher nachzuvollziehen, wenn die unerreichbaren Zustände noch nicht entfernt wurden, was Beispiel 05_demo.xml zeigt, wenn man diesen Epsilon-Automaten in einen NEA umwandeln lässt und nachher die unerreichbaren Zustände entfernt.

Zu beobachten ist, dass lange Epsilon-Ketten unter Umständen viele neue Übergänge ins Leben rufen, wie Beispiel 06_demo.xml exemplarisch zeigt.

f. Automat in einen DEA umwandeln

Jeder beliebige (von meiner Software unterstützte) Automat kann in einen nichtdeterministischen endlichen Automaten (DEA) umgewandelt werden, welcher dieselbe Sprache akzeptiert wie der ursprüngliche Automat. Falls der Ausgangsautomat bereits ein DEA ist, macht der Algorithmus nichts. Falls der Ausgangsautomat kein NEA ist, wird dieser zuerst in einen NEA mit gleicher akzeptierter Sprache transformiert.
Hat der Ausgangsautomat n Zustände, so kann der neue DEA bis zu 2^n Zuständen haben. Unerreichbare Zustände werden jedoch gleich im Transformationsverfahren weggelassen. Mein Algorithmus arbeitet gemäss dem Beweis von Satz 2.2.7 bzw. dem Beispiel 2.2.8 aus dem Skript. Das Ganze befindet sich in der Methode transformToDEA der Klasse Transformation.
Nachdem sichergestellt wurde, dass als Ausgangsautomat nun ein NEA vorliegt, wird zuerst das Alphabet des Automaten abgefragt und anschliessend dessen Zeichen sortiert. Danach wird die erste Zeile der „DEA-Tabelle“ ausgefüllt, analog zu den Übungsserien der Vorlesung.

Weiter wird nun in der Methode toDEAComputeTable Zeile für Zeile abgearbeitet. Dabei können weitere Zeilen der Tabelle hinzugefügt werden. Wenn die letzte Zeile abgearbeitet wird, und in diesem Vorgang keine neue mehr dazukam, werden aus dieser Tabelle die neuen Zustände für den DEA konstruiert. Dabei kommt es zu einer neuen Indexierung, da „Zustands-Zusammenschlüsse“ wie q_1q_2 wie in den Übungsserien von mir als Format nicht unterstützt werden. Dies würde meine Darstellung der Zustände verunmöglichen falls es längere Konkatenationen gäbe.

Als vorletzter Schritt müssen die ganzen Übergänge mit ihren Zeichen aus der Tabelle in den neuen DEA gegossen werden. Als letztes wird lediglich noch der oben besprochene Layout-Algorithmus auf den Automaten angewendet.

Der Algorithmus kann z.B. anhand des Demo-Automaten 01_demo.xml ausprobiert werden. Aus ursprünglich fünf Zuständen entsteht dort ein DEA mit neun Zuständen.
Wie immer finden grafische Aktualisierungen der Darstellung des Automaten statt.

g. Automat in einen minimalen DEA umwandeln

Die Funktionalität wurde wiederum in der Klasse Transformation implementiert, namentlich in der Methode transformToMinimalDEA.
Ausgangsautomat für den Algorithmus ist ein DEA ohne unerreichbare Zustände. Ist der übergebene Automat vom Editor dies nicht, so wird er mittels den oben beschriebenen Verfahren in einen solchen umgewandelt. Es wird ebenfalls wieder auf einer Kopie des Automaten gearbeitet, aus denselben Gründen wie bereits beschrieben.

Der Algorithmus hält sich so nah wie möglich an das in Abbildung 2.3.2 beschriebene Verfahren. Implementiert habe ich dies in der Methode getMinimalDEA der Klasse Transformation. Die Dreieckstabelle aus der Abbildung wird mit einem zweidimensionalen Boolean-Array repräsentiert, von welchem die eine Hälfte nicht gebraucht wird. In diesem Array werden Zustandspaare, welche nicht äquivalent sind, iterativ markiert.

Zuerst werden Paare aus Endzuständen und Nicht-Endzuständen als nicht äquivalent markiert. Dies ist er erste Schritt vor dem Eintritt in die Schleife.

Für die repeat-Schleife aus Abbildung 2.3.1 wird zuerst das Alphabet des Automaten geholt. Nachher wird für nicht markierte Paare, welche durch Lesen eines gleichen Zeichens jeweils zu einem bereits markierten Paar führen, ebenfalls in der Dreieckstabelle markiert (Zeile 8 des Pseudo-codes aus Abbildung 2.3.1). Dies geschieht in der fantasievoll benannten Methode toMinDEAIterationMarking. Da diese über die Dreieckstabelle und zusätzlich noch über das Alphabet iterieren muss, gibt es diverse Verschachtelungen von Kontrollstrukturen. Als Rückgabewert teilt die Methode mit, ob ein neues Zustands-Paar markiert wurde. War dies nicht der Fall, wird es keine weiteren Aufrufe dieser Methode geben seitens der Methode toMinimalDEA.
Damit ist Abbildung 2.3.2 abgeschlossen, und es kann mit der Verschmelzung von äquivalenten Zuständen begonnen werden.

Die Verschmelzung der Zustände aus Abbildung 2.3.3 ist in der Methode toMinDEAStateMerging implementiert. Da die ganzen Übergänge aus dem alten Automaten noch auf die alten Zustände referenzieren, habe ich mich für ein mapping entschieden, welches für jeden Zustand aus dem alten Automaten besagt, in welcher Äquivalenzgruppe des neuen Automaten dieser zu finden ist. Im neuen minimalen DEA wird jede Äquivalenzgruppe zu einem neuen Zustand vermengt.

Als letzter Schritt müssen die Übergänge des alten Automaten in den neuen übertragen werden. Dies geschieht mittels des erstellten mappings. Damit geht’s zurück in die aufrufende Methode transformToMinimalDEA, welche ihrerseits noch kurz schaut ob’s überhaupt äquivalente Zustände gab bzw. ob ein neuer minimaler DEA konstruiert wurde, und die dieses Resultat returniert, was z.B. für die Typbestimmung der Automaten-Information gebraucht wird, ob ein Automat ein minimaler DEA ist. Zu allerletzt wird auch wieder der Layout-Algorithmus aufgerufen.

Die Wirkungsweise des Verfahrens ist gut einsehbar anhand der Beispiele aus der Vorlesung bzw. der Übungsserien, oder auch an meinem eigens kreierten DEA 08_demo.xml.
Alternativ kann man das Transformationsverfahren auch am Epsilon-Automaten 02_demo.xml testen und dann vergleichsweise alle Schritte einzeln selber ausführen und die Resultate vergleichen („Transform to NFA“, „Transform to DFA“, Transform to minimal DFA“).

h. Wort auf Akzeptanz testen

Der Benutzer kann testen, ob ein bestimmtes Wort vom Automaten akzeptiert wird, indem er im Automaton-Menü auf den Eintrag „Word accepted“ klickt. Nach Eingabe des Wortes folgt eine Animation, welche aufzeigt, wie der Automat das Wort verarbeitet. Falls das Wort akzeptiert wird, geht die Animation so lange bis alle Zeichen gelesen sind und man beim Endzustand angekommen ist. Es wird immer nur eine Lösung abgespielt, auch wenn der Automat das Wort auf verschiedene Arten akzeptieren könnte.
Wird das Wort vom Automaten nicht akzeptiert, wird die Animation des längsten, akzeptierten Teilwortes ab dem ersten Zeichen abgespielt.

Um sich einen Eindruck über die Animation zu verschaffen, eignet sich z.B. der Demo-Automat 02_demo.xml mit den zu testenden Wörtern „aabcabaca“ und „abac“.

Der Test auf die Akzeptanz des Wortes findet in der Methode readSymbol der Klasse Language statt. Falls das Wort akzeptiert wird, tritt der Basisfall der Rekursion ein. Im Falle der Nicht-Akzeptanz werden weiter unten irgendwann keine rekursiven Aufrufe mehr folgen. Die Funktionsweise des Algorithmus ist wieder ähnlich wie bei den bereits besprochenen Algorithmen. Speziell ist lediglich, dass getestet werden muss, dass die Methode sich nicht im Kreis herum selber aufruft während sie ständig leere Wörter liest. Ein Stack Overflow wäre die Folge.

Ursprünglich wollte ich dieses Problem lösen, indem ich einfach „Epsilon-Kreise“ aus dem Automaten entferne. Dies kann jedoch die akzeptierte Sprache des Automaten verändern, und ist demzufolge inakzeptabel. Die Lösung war eine Überprüfung, ob man durch das Traversieren des bevorstehenden Epsilon-Übergangs damit einen Epsilon-Kreis schliesst (man betrachtet die Historie der zuvor gelesenen Zeichen). Falls dem so wäre, springt man in der Schleife zur nächsten Iteration.

Ebenso darf der Algorithmus noch nicht terminieren, wenn alle Zeichen gelesen sind, man sich jedoch noch nicht in einem Endzustand befindet. Denn man könnte je nach Automat durch traversieren weiterer Epsilon-Übergängen noch in einen Endzustand gelangen, womit das Wort doch noch akzeptiert würde.

Aufbau der Software (Klassenübersicht)

- editor-package
    - Editor

      In der Klasse Editor befindet sich die main-Methode. Diese ruft die klasseneigene run-Methode auf, welche eine Endlosschleife enthält, die in der momentanen Konfiguration 30 Durchläufe pro Sekunde absolviert, vergleichbar mit den „frames per second“ eines Videospiels. Da ich auch ein paar Animationen habe und der Benutzer Elemente des Automaten mit der Maus verschieben kann, erschien mir dies ein guter Ansatz. Es ist natürlich deshalb eine Endlosschleife, da der Benutzer das Programm selber wie jedes andere Programm beenden kann. In jedem Schleifendurchlauf finden bspw. grafische Aktualisierungen statt.
      Die Editor-Klasse erbt von Javas JFrame-Klasse. Meine Klasse hat im Wesentlichen ein Objekt der Klasse Toolbar (die linke Werkzeugleiste mit den Zustands-buttons) und ein Objekt der Klasse DrawablePanel, welche die „Zeichnungsfläche“ für den Automaten bereitstellt.

    - DrawablePanel

      Die „Zeichnungsfläche“ für den Automaten, Subklasse der von Javas JPanel-Klasse. Diese Klasse implementiert das MouseMotionListener-Interface. Für die weiteren Abfragen des Benutzer-inputs wurde ein KeyListener und ein MouseListener hinzugefügt. Dies erlaubt mir diverse Abfragen über den Maus- und Tastaturstatus.
      Die Klasse enthält das Objekt der Klasse Automat, also all relevanten Informationen zu dem Automaten selber. Ich habe mich im Nachhinein gefragt ob diese Objekt wirklich da hingehört und bin mir nach wie vor nicht

wirklich sicher. Gestört hat mich dieser Aufbau bei der Entwicklung jedoch nie.

- Toolbar

  Diese Klasse enthält die buttons der Werkzeugleiste am linken Rand des GUI. Wie bei allen grafischen Elementen (Klassen, welche von der Klasse JComponent geerbt haben) überschreibt sie deren paint-Methode.

- Automat

  Die wohl wichtigste Klasse. Sie enthält die relevanten Informationen zu dem Automaten, bspw. die Zustände und die Übergänge. Da diese Klasse Gefahr lief zu gross zu werden, habe ich mich bemüht, Sachen welche nicht zwingend da hingehörten, auszulagern. Die ganzen Transformations-Methoden „NEA nach DEA“ etc. hätten ja eigentlich auch in die Klasse Automat gepasst, aber ich war mit der Auslagerung von solcher und ähnlicher Funktionalität nicht unglücklich.
  Die Klasse Automat hat eine ArrayList von den Zuständen. Die Indexierung der Zustände des Automaten ist jedoch nicht dieselbe wie diejenige der ArrayList selber. Beim Löschen eines Zustands aus der ArrayList würde die Indexierung die Lücke schliessen. Ein Verhalten, was ich bei den Automaten-Zuständen nicht will.
  Ebenso enthält die Klasse eine ArrayList mit den Übergängen (Klasse Transition).
  Ausserdem finden viele Vorgänge wie bspw. die Konstruktion eines neuen Überganges, die input-Verarbeitung bei Mausklicks oder das Löschen eines Zustandes mit seinen Folgen hier statt.

- State

  Repräsentiert einen Zustand. Von dieser Klasse erben die Klassen StartState, EndState und die Klasse StartEndState (Java kennt keine Mehrfach-Vererbung wie bspw. C++). Informationen, welche alle Zustände enthalten finden sich dementsprechend hier, z.B. der Zustands-Index oder die x-y-Koordinaten des Zustandes.
  Die Klasse State erbt wiederum von der Klasse Shape.

- Shape

  Eine abstrakte Klasse, welche als Superklasse der Klassen State, Transition und Symbol dient. Diese verlangt bei der Vererbung die Implementierung der Methode mouseClickHit, welche festlegt, was bei einem Maus-Klick auf den Zustand bzw. den Übergang oder das Übergangssymbol geschehen soll. Ausserdem enthält sie einen flag der angibt, ob ein Element grafisch markiert ist.

- Transition

  Repräsentiert einen Übergang des Automaten. Dieser enthält eine Referenz auf den Startzustand und den Endzustand des Übergangs. Ebenso enthält die Klasse eine ArrayList von Objekten der Klasse Symbol (ein Symbol eines Übergangs).
  Ich habe mich dazu entschieden, die grafischen Informationen und Berechnungen der Übergänge auszulagern, da sonst die Klasse auch wieder zu gross geworden wäre. Somit enthält die Klasse Transition ein Objekt der abstrakten Klasse TransitionPaint, welche als Superklasse der Klassen TransitionPaintLine (ein „gerader“ Übergang zwischen zwei verschiedenen Zuständen) und der Klasse TransitionPaintArc (ein Übergang zwischen demselben Zustand, grafisch repräsentiert durch einen Bogen).

- XMLFileParser

  Stellt die Funktionalität für das Speichern und Laden eines Automaten bereit.

- controlFlow-package

  - UserAction

    Das controlFlow-package enthält lediglich die undo-redo Funktionalität. Herzstück davon ist die abstrakte Klasse UserAction. Diese dient als Superklasse für die Klassen AddedState, RemovedTransition etc. Die Klasse UserAction enthält eine Liste der letzten Aktionen des Benutzers. Diese Liste wird traversiert falls der Benutzer den undo- bzw. redo-Menüeintrag auswählt. Ebenso wird beim Traversieren die ensprechende Aktion aufgerufen, damit der Benutzer die Veränderung des Automaten auch optisch sieht. Hierzu dienen die abstrakten Methoden undo und redo. Erbende Klassen müssen diese implementieren und damit festlegen, was bei einem undo bzw. redo passieren soll. Bei einem undo löscht z.B. ein Objekt der Klasse AddedState den hinzugefügten Zustand wieder. Diese kleine Klassenhierarchie erlaubt mir, eine LinkedList mit dem generischen Typ UserAction zu verwenden (zu finden in der Klasse UserAction). Beim Aufruf von undo bzw. redo greift dann der Polymorphismus und wählt die spezifische Methode des Objektes auf.

- transformation-package

    - Util

      Das transformation-package enthält im Wesentlichen die Funktionalität für den dritten Menueintrag „Automaton“. Funktionalität, welche von mehreren Menüpunkten gebraucht wird findet man in der Klasse Util. Man hätte praktisch das gesamte package auch in die Klasse Automat platzieren können. Ich entschied mich dann aber doch für ein eigenes package mit fast ausschliesslich statischen Methoden, da ohnehin keine Objekte davon erzeugt werden müssen. Von dieser Design-Entscheidung bin ich nicht so richtig überzeugt, allerdings hätte ich auch heute nicht eine bessere Idee auf Lager. Mittels der Util-Klasse kann das Alphabet des Automaten abgefragt werden, von einem Zustand die Nachbarzustände ermittelt werden, Abfragen durchgeführt werden bzgl. des Automaten-Typs, die äquivalente Grammatik des Automaten ermittelt werden etc.

    - Transformation

      Die Klasse Transformation enthält die Umwandlungs-Algorithmen der Menüpunkte „Transform to NFA“, „Transform to DFA“ und „Transform to minimal DFA“. Dadurch ist eine ganze Menge von privaten Hilfsmethoden nötig geworden. Diese Transformationen von Automaten-Typen orientieren sich streng nach den Vorlagen des Vorlesungsskripts, auf welches am Anfang dieser Dokumentation verwiesen wurde.

- Layout

  In dieser Klasse befindet sich die Funktionalität des Menüeintrages „Layout“. Die Methode layoutAutomat führt dabei die in Punkt 5c aufgeführten Schritte aus.

- Language

  Diese Klasse kann testen, ob der Automat ein Wort akzeptiert. Ausgangspunkt ist die dabei die Methode wordAccepted. Diese führt die in Punkt 5h erläuterten Punkte aus.

„Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist."

Bern, 20. 4. 2017

Raphael Bucher

Raphael Bucher