

HOCHSCHULE ZITTAU/GRLITZ (FH)
FACHBEREICH INFORMATIK

Entwicklung eines XML-basierten Editors für formale Automaten in Delphi mit publikationsfreundlichen Ausgabeformaten

Praxissemesterarbeit

der Hochschule Zittau/Grlitz (FH)

vorgelegt von Michael Hielscher
geboren am 01. August 1982 in Zittau.

Betreuer: Prof. Dr. Christan Wagenknecht

Görlitz, 11.03.2005

Inhaltsverzeichnis

1	Einleitung	3
2	Aufgabenstellung	4
3	Vorhandene Software und ihre Grenzen	5
3.1	JFLAP - Java Formal Languages and Automata Package	6
3.2	DEM - Deus Ex Machina	7
3.3	FSMS and PDAS Finite State Machine Simulator and Push-Down Automaton Simulator	8
4	Theoretische Grundlagen	9
4.1	Automatentheorie	9
4.2	XML und XML Schemas	11
4.3	SVG Scaleable Vector Graphics	14
5	Entwicklung von AutoEdit	15
5.1	XML zur Definition eines abstrakten Automaten	15
5.2	Zerlegung in Teilprobleme	17
5.3	Klassen	18
6	Programmierung mit Delphi	21
6.1	Verarbeitung von XML mit MSXML 4.0	21
6.2	Ausgesuchte Codebeispiele	24
6.2.1	Überprüfung eines Eingabewortes	24
6.2.2	Snapping für den visuellen Entwurf	25
6.2.3	Minimieren eines NEA's	27
6.2.4	Generieren von Scheme Code für DEA	29
6.3	Anbindung an externe Tools	31

7 Weiterentwicklungsmöglichkeiten	32
8 Fazit	33
9 Anhang	34
9.1 XML Schema für Automaten	34
9.2 Tutorial	43
9.3 Datenträgerbeschreibung	51
10 Literaturverzeichnis	52

1 Einleitung

Unterrichtslehrpläne diverser Bundesländer enthalten in der Jahrgangsstufe 11/12, den Lernbereich theoretische Informatik, mit dem Schwerpunkt „Theoretische Grundlagen von Programmiersprachen“, welcher sich vorwiegend mit Sprachen und Automaten beschäftigt.

Die Hauptinhalte sind dabei reguläre und kontextfreie Sprachen, endliche Automaten, Kellerautomaten und Turingmaschinen. Der Aufbau und die Arbeitsweise dieser Automaten sollen mit einfachen Beispielen veranschaulicht werden.

Von Lehrerinnen und Lehrern wird heute zunehmend erwartet, dass sie schriftliche Unterrichtsmaterialien, wie Arbeitsblätter, Lehrtexte, Übungsaufgaben und Klausuren als Textdokumente am Computer produzieren. Diese werden ausgedruckt bzw. ins Web gestellt.

Für InformatiklehrerInnen, die mit Computeranwendungen der entsprechenden Art vertraut sind, ist die Herstellung solcher Materialien dennoch ein großes Zeitproblem, denn die typografischen Anforderungen an die Texte sind hoch: Schüler begnügen sich nicht mit reinem Textformat im E-Mail-Stil, sondern erwarten typografisch anspruchsvolle Dokumente, die oftmals Formeln und grafische Darstellungen enthalten. Gängige Textprozessoren verfügen über Formeditoren und Zeichenkomponenten, die den „Lehrmaterial-Redakteur“ unterstützen.

Dennoch gibt es Felder, in denen LehrerInnen bisher vergeblich nach einem geeigneten Werkzeug suchen. Beispielsweise fehlt es an einem Editor für Graphen abstrakter Automaten. Ein solcher Editor sollte die Definition abstrakter Automaten ermöglichen, zugehörige Zustandsüberführungsgraphen generieren, bzw. automatisch anordnen und manuelle Nachbesserungen gestatten. Hinzu kommt, dass Lehrende unmittelbar beim Entwurf eines Beispielautomaten dessen Anwendung auf Eingabewörter erproben bzw. simulieren möchten. Folglich sollte ein Entwurfswerkzeug für abstrakte Automaten nicht nur ein Editor sein, der gewisse Publikationsaufgaben unterstützt, sondern sollte darüber hinaus Simulationen und gängige Transformationen ermöglichen. Im übertragenen Sinne ist ein dementsprechendes Werkzeug auch für die Hand des Schülers geeignet.

2 Aufgabenstellung

Für die anschauliche Repräsentation abstrakter Automaten, wie aus der theoretischen Informatik bekannt, werden Zustandsgraphen verwendet.

Automaten werden unter anderem, durch eine Überföhrungsfunktionen beschreiben. Diese kann man sowohl als Graph, als Tabelle oder auch in der Form: $\delta(\text{Zeichen}, \text{Zustand}_1) \Rightarrow \text{Zustand}_2$ darstellen.

Für Lehrmittel (z.B.: Vorlesungsscripte, Arbeitsblätter) auf Papier wie auch im Internet, soll eine halbautomatische Publikationsumgebung für abstrakte Automaten entstehen.

Für die Automaten soll dabei eine XML basierte Repräsentation gewählt werden, um ein möglichst vielseitiges Format für zukünftige Umformungen oder Ähnliches zu gewährleisten. Dabei sollen die verschiedenen Typen (DEA, NEA, DKA, NKA, TM) zusammen mit entsprechenden Transformationen implementiert werden.

Um die benötigten Graphen zu erstellen wird eine Manipulationsumgebung benötigt. Dabei soll es möglich sein, die einzelnen Elemente einfach mit der Maus zu bewegen, bis die Anordnung den Wünschen des Autors entspricht.

Eine vollautomatische Publikation wird nicht angestrebt.

3 Vorhandene Software und ihre Grenzen

Auf dem Markt findet man bereits gute Softwarelösungen für die einzelnen Teilgebiete, wie Simulation, Transformation (zum Beispiel JFLAP) und Präsentation (zum Beispiel MS Visio). Dennoch findet man keine Gesamtlösung, die all diese Gebiete in einem Programm vereint. Durch ein fehlendes gemeinsames Datenformat ist der Austausch von Automatendefinitionen zwischen den vorhandenen Programmen auch sehr schwierig.

Für Autoren, die ihre Materialien mit \LaTeX verfassen, gibt es auch direkte Packages für das Zeichnen von Zustandsübergangsgraphen. Diese beschränken sich jedoch zum einen ausschließlich auf die Darstellungsaufgabe und zum anderen müssen diese auch in Kommandoform notiert werden, was umfangreiche Kenntnisse über das verwendete Package erfordert. Der fertige Graph wird erst nach dem Compilieren des \LaTeX -Dokuments sichtbar und kann somit nicht interaktiv bearbeitet werden. (Ein Beispiel für ein solches Package: [Vaucanson-G].)

Bildverarbeitungssoftware, wie etwa MS Visio, bietet die Möglichkeit, Graphen zu erstellen und in für Printmedien sinnvollen Auflösungen zu speichern. Der große Nachteil ist jedoch der hohe Zeitaufwand, da der Anwender bei der Anordnung und Ausrichtung kaum unterstützt wird. Weiterhin ist es nicht möglich, den gezeichneten Automaten zu simulieren oder zu transformieren. Eine Weiterverarbeitung in einem entsprechenden Tool ist ebenfalls unmöglich.

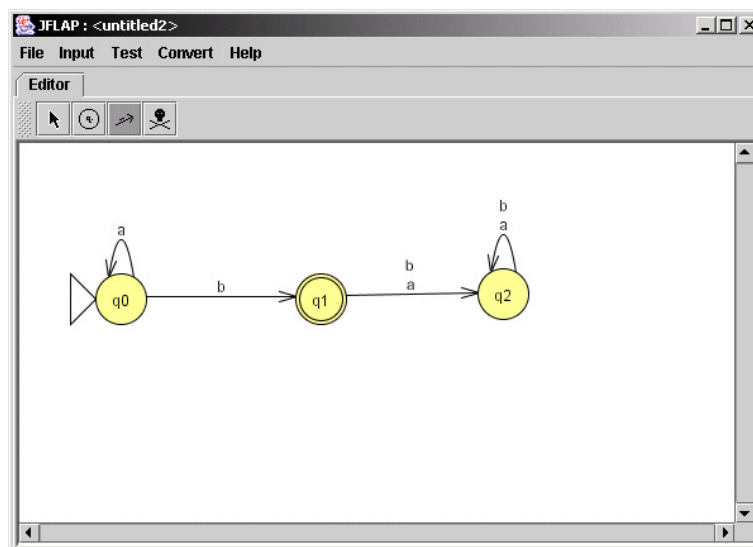
Tools, wie JFLAP, bieten eine Vielzahl von Transformations- und Simulationsmöglichkeiten, unterstützen den Anwender bei der Erstellung von Zustandsübergangsgraphen und erlauben somit ein schnelles Zeichnen und Testen von Automaten. Die Präsentation beschränkt sich aber auf den Bildschirm und es wird derzeit keine Möglichkeit geboten, einen erstellten Graphen in ein für Printmedien geeignetes Format zu exportieren. ([FLAT] bietet eine Liste von derzeit vorhandenen Tools.)

Im Folgenden sollen aktuelle Softwarelösungen kurz vorgestellt werden.

3.1 JFLAP - Java Formal Languages and Automata Package

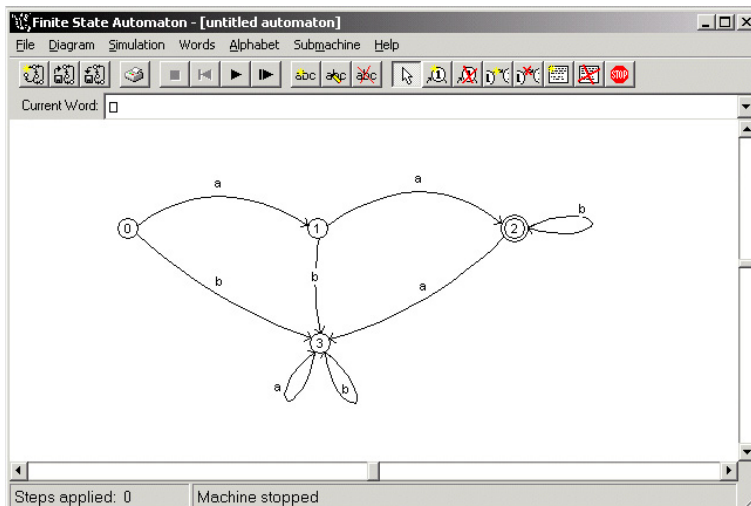
JFLAP ist das derzeit bekannteste und umfangreichste Tool zu dieser Thematik. Es eignet sich gut zur Simulation und Transformation von Automaten, erlaubt jedoch keine benutzerdefinierten Einstellungen für die Darstellung der Zustandsübergangsgraphen (wie Farben oder Größen). Des Weiteren ist die Druckoption nicht so umfangreich gestaltet, als das man diese für Printmedien nutzen könnte.

JFLAP bietet außerdem umfangreiche Funktionen zur Transformation von Grammatiken. Die Benutzungsoberfläche ist außerdem wohl die übersichtlichste von den hier vorgestellten Tools.



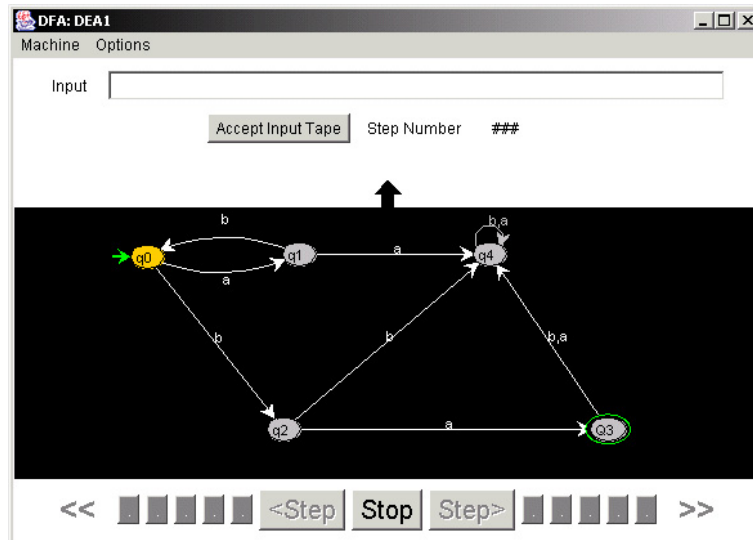
3.2 DEM - Deus Ex Machina

Dieses Tool konzentriert sich praktisch ausschließlich auf die Simulation und legt nicht zu viel Wert auf die graphische Darstellung. Damit ist die Nutzung in Dokumenten unsinnig. Der Funktionsumfang ist im Vergleich zu JFLAP geringer. Ein weiterer Nachteil ist die Aufspaltung der Software in viele Einzelteile für jeweils einen Automatentyp. Dies ist für den Nutzer teilweise verwirrend und erlaubt auch keine Transformationen zwischen Automatentypen.



3.3 FSMS and PDAS Finite State Machine Simulator and Push-Down Automaton Simulator

Dieses Tool beschränkt sich ebenfalls nur auf die Simulation. Es können sowohl endliche Automaten (DEA / NEA) als auch nichtdeterministische Kellerautomaten simuliert werden. Die graphische Darstellung eignet sich aber kaum zum Drucken oder zur Weiterverarbeitung.



4 Theoretische Grundlagen

4.1 Automatentheorie

Aus Sicht der theoretischen Informatik, sind Automaten abstrakte Maschinen. Diese Maschinen bestehen im Wesentlichen aus einer endlichen Anzahl von Zuständen und einer Überföhrungsfunktion, welche darüber bestimmt unter welchen Bedingungen von einem Zustand in einen Anderen gewechselt wird. Einer dieser Zustände wird als Startzustand gewöhlt. Desweiteren wird ein oder mehrere Zustände als Endzustand deklariert (Startzustand kann dabei auch gleichzeitig ein Endzustand sein). Darüber hinaus benötigt eine solche Maschine ein Eingabewort welches sie bearbeiten kann. Dieses Wort besteht wiederum aus Zeichen des Eingabealphabets, mit dem der Automat arbeitet. Dabei sind die einzelnen Zeichen nicht zwangsweise identisch mit denen der natürlichen Sprache. Es ist durchaus denkbar, dass ein einziges Zeichen des Eingabealphabets einem ganzen Wort der natürlichen Sprache entspricht (z.B.: BEGIN, END).

In der theoretischen Informatik werden Sprachen durch eine Anzahl von Regeln definiert: einer so genannten Grammatik. Je nach Form der Regeln teilt man Grammatiken, nach Chomsky¹, in vier Klassen ein.

Typ 0: unbeschränkt ohne jegliche Einschränkungen

Typ 1: kontextsensitive Grammatiken

Typ 2: kontextfreie Grammatiken

Typ 3: reguläre Grammatiken

Äquivalent dazu benennt man die entsprechenden Sprachklassen unbeschränkt, kontextsensitiv, kontextfrei bzw. regulär. Dabei beinhalten die Typ 2 Sprachen alle Typ 3 Sprachen, Typ 1 Sprachen alle Typ 2 und Typ 3 Sprachen usw.

Entsprechend dazu, kann man verschiedene Typen von Automaten unterscheiden. Deter-

¹Noam Chomsky, geb.1928

ministische endliche Automaten (DEA) und nichtdeterministische endliche Automaten (NEA) entsprechen im Repräsentationsumfang den regulären Sprachen (Chomsky Typ 3). Durch Kellerautomaten (DKA bzw. NKA) wird es möglich, bis einschließlich Typ 2 Sprachen zu erfassen. Für Typ 0 und Typ 1 Sprachen benötigt man Turing Maschinen (kurz TM) oder Registermaschinen. Durch verschiedene Verfahren ist es möglich, Automaten eines bestimmten Typs in einen anderen Typ umzuwandeln. Dies ist aber nur zwischen Automatentypen des gleichen Rangs möglich. Es ist demnach unmöglich, einen Kellerautomaten in einen nichtdeterministischen Automaten umzuwandeln (aber durchaus umgekehrt).

4.2 XML und XML Schemas

XML ist eine einfache Textsprache, die durch ihre große Flexibilität in praktisch allen Bereichen eingesetzt werden kann, wo Informationen in Dateien gespeichert werden müssen.

XML ist eine markup language ähnlich wie HTML, wobei die Zielsetzung beider Sprachen unterschiedlich ist. HTML konzentriert sich auf die Beschreibung der Darstellung von Informationen, wohingegen XML auf die Beschreibung der Information selbst ausgelegt ist. XML wurde auch nicht dafür entwickelt irgendeiner konkreten Funktion gerecht zu werden. Es geht einfach und allein um die Strukturierung und Aufbewahrung von Daten jeglicher Art.

In XML gibt es also keine vorgegebenen Tags wie `<BODY>` oder `<TABLE>`. Tags müssen erst selbst definiert werden, wodurch auch das Wort "Extensible" (Erweiterbar) gerechtfertigt ist. Diese Definition geschieht über DTD (Document Type Definition) oder über die Weiterentwicklung, den XML-Schema Dateien.

Da XML-Schemas selbst dem XML Standard entsprechen und eine größere Funktionalität aufweisen, werden diese wohl zukünftig die größere Rolle spielen.

Ein Beispiel für ein solches Schema:

```
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">

  <xs:element name="Kunde">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Vorname" type="xs:string"/>
        <xs:element name="Adresse" type="xs:string"/>
        <xs:element name="Geburtsdatum" type="xs:date"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Heute gibt es bereits viele Editoren die es erlauben, ein solches Schema graphisch zu erstellen. Abbildung 4.1 (Seite 12) zeigt eine mögliche Darstellung des obigen Beispiels.

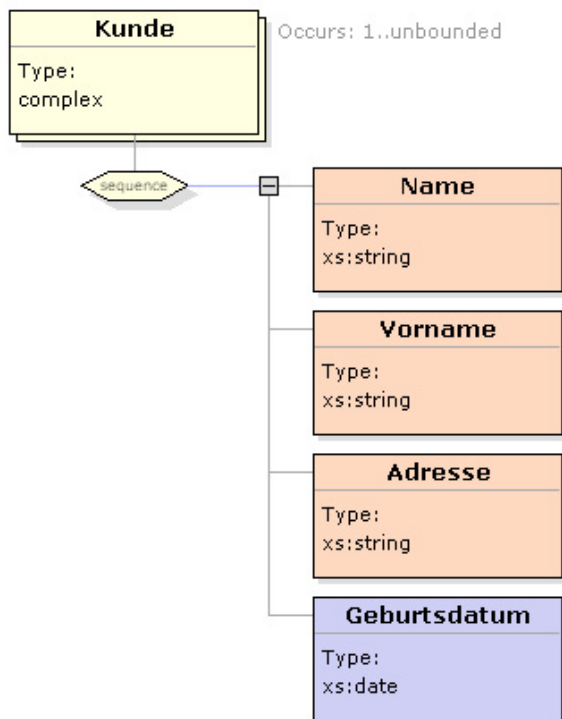


Abbildung 4.1: XML Schema in graphischer Darstellung

Eine passende XML Datei für dieses Schema könnte nun wie folgt aussehen:

```
<Kunde>
  <Name>Mustermann</Name>
  <Vorname>Hans</Vorname>
  <Adresse>12345 Musterhausen</Adresse>
  <Geburtsdatum>2001-01-01</Geburtsdatum>
</Kunde>
```

Wie man erkennt, beinhaltet das XML Dokument nur die Informationen eingefasst in selbstdefinierte Tags. Um diese Daten einem Nutzer präsentieren zu können wird eine Software benötigt, die die XML Daten beispielsweise in HTML überführt, welche dann von jedem handelsüblichen Browser betrachtet werden kann. Diese Transformationen

werden bevorzugt durch eine weitere Sprache XSLT (Extensible Stylesheet Language for Transformations) beschrieben.

Um eine hohe Datenintegrität ähnlich einer Datenbank zu gewährleisten, besitzen Schemas eine breite Palette von Sprachelementen. Von einfachen Stelligkeitsbegrenzungen bis hin zu Patternmatching mit Hilfe regulärer Ausdrücke ist alles möglich, um eine möglichst exakte Beschreibung der Daten festzulegen. XML Dateien können auf Gültigkeit mit einem solchen Schema überprüft werden. Fehlerhafte Eingaben können somit bereits auf dieser Ebene ausgewertet und gegebenenfalls korregiert werden. Eine Vielzahl von vorgefertigten Variablentypen wie `xs:string`, `xs:integer` ... erlauben eine schnelle Festlegung des Typs der zu erwartenden Daten, welche anschließend noch mit weiteren Attributen zusätzlich eingeschränkt werden können. Für eine Postleitzahl könnte man zum Beispiel folgendes verwenden:

```
<xs:element name="PLZ">
  <xs:simpleType>
    <xs:restriction base="xs:unsignedInt">
      <xs:totalDigits value="5"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Zu beachten bleibt, dass diese Typen, obwohl sie an Programmiersprachen erinnern, stets als Text notiert werden. Ein Integer oder Float Wert wird nicht etwa Binär sondern durch einen ASCII String beschrieben. (Beispiel: `<Steuer>0.16</Steuer>`)

Dies ist aber einleuchtend, da XML wie bereits erwähnt eine Textsprache ist. Davon ausgehend, sollten in XML Dateien keine Binärdaten verwendet werden. Wenn dies dennoch nötig ist, bedient man sich häufig der Base64 Verschlüsselung, die auch für Emailanhänge genutzt wird, um Binärdaten in die ASCII Form zu überführen.

4.3 SVG Scaleable Vector Graphics

Eines der verwendeten Grafikformate in AutoEdit ist SVG. Im Gegensatz zu Bitmapformaten wie GIF oder JPEG werden bei Vektorformaten nicht die Farbdaten der Pixel, sondern die Anweisungen für den virtuellen Stift gespeichert der das Bild bei Bedarf zeichnet.

Dabei werden die Anweisungen in einer SVG Datei Schritt für Schritt abgearbeitet um letztendlich das gewünschte Bild zu erzeugen. Daraus ergeben sich einige Vorteile zu Bitmapformaten. Der für AutoEdit bedeutenste Vorteil ist die Möglichkeit derartige Graphiken beliebig ohne Qualitätsverlust zu skalieren womit zu jedem Zeitpunkt eine optimale Druckqualität erreicht wird. Durch die Repräsentation als Textanweisungen ist es natürlich auch möglich Inhalte einzufügen oder zu entfernen, was bei Bitmaps praktisch unmöglich ist.

Dennoch ist der Einsatz von Vektorformaten nicht immer sinnvoll. Die Darstellung einer Photographie ist beispielsweise unmöglich.

SVG ist ein Vektorformat bei dem die Befehle XML gerecht gespeichert werden. Um ein Rechteck zu zeichnen wird beispielsweise ein solcher Befehl verwendet:

```
<rect x="400" y="100" width="400" height="200"
      fill="yellow" stroke="navy" stroke-width="10" />
```

Somit ist SVG ein Anwendungsbeispiel für die Vielseitigkeit von XML. Im Vergleich ein anderes Vektorformat - Postscript:

```
newpath
0.000 0.000 0.000 setrgbcolor
2 setlinewidth
0 0 moveto
0 50 lineto
50 50 lineto
50 0 lineto
closepath
stroke
```

Diese Zeilen führen in Postscript zur Darstellung eines schwarzen Rechtecks jedoch sind die Anweisungen deutlich schwerer lesbar und weniger intuitiv.

5 Entwicklung von AutoEdit

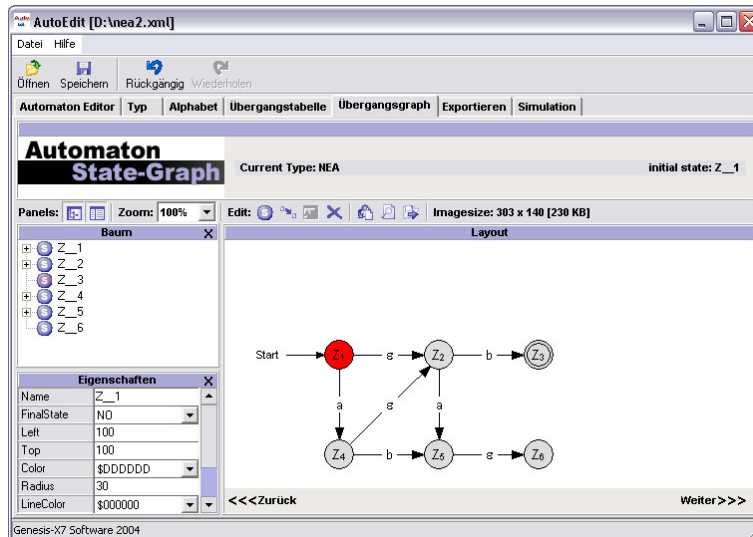


Abbildung 5.1: AutoEdit Screenshot

5.1 XML zur Definition eines abstrakten Automaten

Ausgehend von den vorangehenden Betrachtungen, ist es von großem Vorteil die Definition eines abstrakten Automaten XML gerecht zu speichern.

Zunächst müssen allgemeine Daten über einen Automaten erfasst werden. Diese betreffen zum einen den Typ (DEA, NEA ...) und zum anderen das zu verwendende Alphabet, Startzustand und ähnliche globale Informationen. Vorerst kann man diese Überlegungen intuitiv in XML niederschreiben:

```
<AUTOMATON>
  <TYPE value="DEA"/>
  <ALPHABET>
    <ITEM value="a"/>
```

```

    <ITEM value="b"/>
    <ITEM value="c"/>
  </ALPHABET>
  <INITIALSTATE value="Z1"/>
</AUTOMATON>

```

Zu diesen statischen Informationen benötigt man natürlich die Zustände des Automaten und die Übergänge zwischen den Zuständen. Obiger Code erweitert mit zwei Beispiel Zuständen und einem Übergang zwischen Z1 nach Z2. Jeder Zustand besitzt ein boolisches Attribut für Endzustand, ein Übergang besitzt ein Ziel (target) und ein LABEL bezeichnet jeweils ein Alphabetszeichen, mit welchem der übergeordnete Übergang möglich ist.:

```

<AUTOMATON>
  <TYPE value="DEA"/>
  <ALPHABET>
    <ITEM value="a"/>
    <ITEM value="b"/>
    <ITEM value="c"/>
  </ALPHABET>
  <INITIALSTATE value="Z1"/>
  <STATE name="Z1" terminalstate="false">
    <TRANSITION target="Z2">
      <LABEL read="a"/>
      <LABEL read="b"/>
      <LABEL read="c"/>
    </TRANSITION>
  </STATE>
  <STATE name="Z2" terminalstate="false"/>
</AUTOMATON>

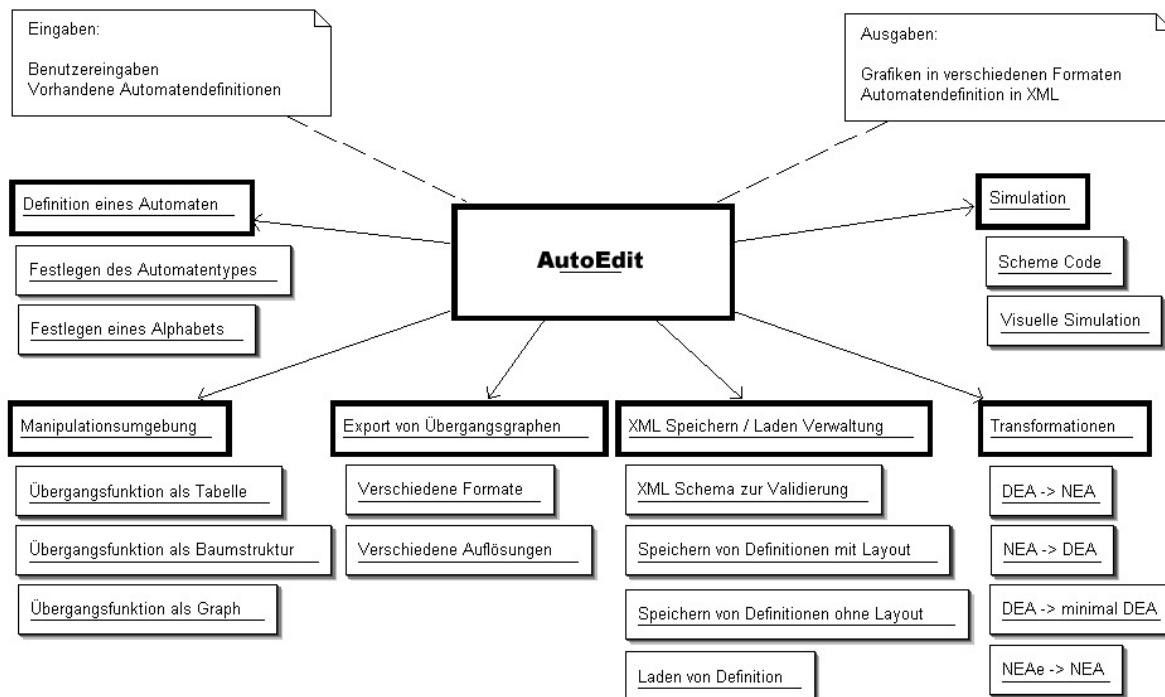
```

Ob man die Informationen, wie oben gezeigt, als Parameter an die Tags bindet oder `<TYPE>DEA</TYPE>` schreibt ist nicht festgelegt. Wenn man aber eine der beiden Möglichkeiten konsequent benutzen, erhält man eine etwas leichter lesbare (parsebare) Definition.

Für dieses XML Format wurde ein XML Schema definiert. Dieses kann im Anhang nachgelesen werden.

5.2 Zerlegung in Teilprobleme

Ausgehend von der Aufgabenstellung wurde ein Systementwurf erstellt welcher das Gesamtproblem in Teilprobleme zerlegt.



Dieser Entwurf ist jedoch nicht vollständig, da das Projekt im Laufe der Entwicklung um neue Funktionalität erweitert wurde.

Aus den Teilaufgaben wurden Klassen für eine objektorientierte Entwicklung abgeleitet. Dabei wurde das MVC Paradigma angewendet. Durch die visuelle Entwicklung mit Delphi ist sowohl der Teil View als auch Control weniger von Bedeutung. Für die Benutzungsoberfläche wurden wiederverwendbare Komponenten entwickelt um einen einheitlichen Stil zu gewährleisten. Die Steuerung wird von Delphi selbst übernommen und nur die entsprechenden Ereignisse mit Methoden aus dem Model verbunden.

Deshalb wird im Folgenden nur noch der Modelteil betrachtet.

5.3 Klassen

Die Hauptklasse innerhalb von AutoEdit ist TAutomaton. In dieser liegt eine Datenstruktur AAutomaton welche zur Laufzeit alle Daten des in Bearbeitung befindlichen Automaten beinhaltet. Dies ist die einzige Stelle im Speicher wo die Zustände, Übergänge und Labels gespeichert sind. Alle anderen Klassen die diese Informationen benötigen greifen auf AAutomaton zurück.

TAutomaton bietet eine Vielzahl von Methoden die es erlauben die Daten von AAutomaton zu verändern oder zu durchsuchen. Alle weiteren Klassen des Modelteils sind in Form von Instanzvariablen an die Hauptklasse gebunden. Im nachfolgenden sollen diese kurz beschrieben werden.

TAUTOMATONXML

Alle Aktionen zum Speichern und Laden von Automatendefinitionen in/aus einer XML Datei werden von diese Klasse übernommen. Zu Testzwecken wurde auch eine LoadFromFileJFLAP entwickelt die in der Lage sein soll, daß JFLAP XML Format für DEA und NEA zu lesen. Beim Laden eines AutoEdit Automaten wird das XML Schema angewendet welches im Anhang zu finden ist.

TAUTOMATONSCHEMCODE

Eine der Simulationsmöglichkeiten ist die Erzeugung von Scheme Quellcode für den entwickelten Automaten. Diese Klasse übernimmt das Generieren dieses Quellcodes für alle von AutoEdit unterstützten Automatentypen.

TAUTOMATONTRANSFORMATION

Diese Klasse übernimmt alle Transformationen von einem Automatentyp in einen anderen. Dabei wird bei bestimmten Transformationen die Zustandsübergangstabelle benötigt. Diese wird ebenfalls von dieser Klasse im Speicher erzeugt und wird auch von der Klasse TAutomatonGraph für die Darstellung dieser Tabelle verwendet.

TAUTOMATONVALIDATER

AutoEdit erlaubt die Überprüfung des aktuellen Automaten auf Gültigkeit. Für jeden Automatentyp bietet diese Klasse eine Funktion zur Überprüfung an. Diese Funktionalität beschränkt sich jedoch auf einige ausgesuchte Überprüfungen und kann gerade bei Kellerautomaten nicht als Garantie für die Gültigkeit angesehen werden.

TAUTOMATONSIMULATION

Diese Klasse ist für die visuelle Simulation in AutoEdit verantwortlich. Der animierte Zustandsgraph wird dabei über die Klasse TAutomatonGraph verwirklicht. Die Zustandstabelle auf der Simulationsseite von AutoEdit wird ebenfalls von dieser Klasse erstellt und gezeichnet.

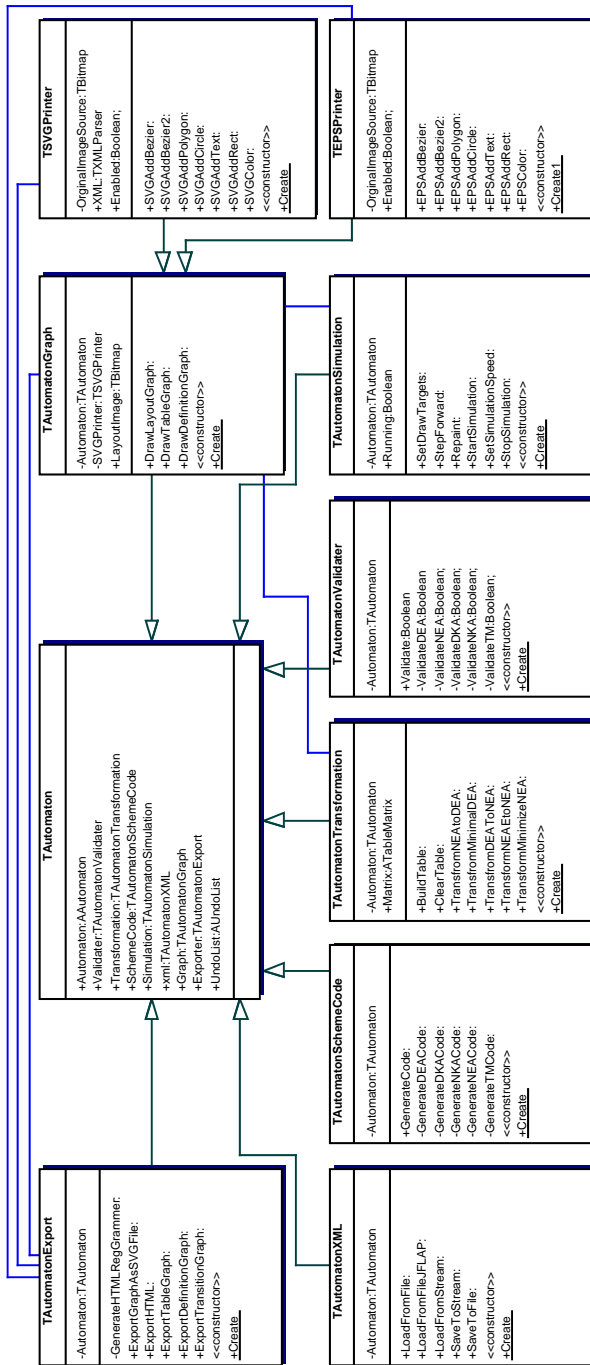
TAUTOMATONEXPORT

Alle Exporte sowohl von Graphen als auch von HTML Dateien oder ähnlichen wird von dieser Klasse erledigt. Für die verschiedenen Bildquellen wird erneut die Klasse TAutomatonGraph verwendet.

TAUTOMATONGRAPH

Alle Graphen werden von dieser Klasse gezeichnet. Ein Bitmap innerhalb dieser Klasse wird zum zeichnen verwendet und anschließend von anderen Klassen als Quelle benutzt. Die Klasse kann Zustandsübergangsgraphen als auch Zustandstabellen und Automaten definitionen als Bitmaps erzeugen. Für Vektorformate gibt es TSVGPrinter und TEPSPrinter. Diese laufen beim Renderingprozess mit und schreiben (wenn aktiv) die entsprechenden Anweisungen in eine virtuelle Datei mit. Diese können dann anschließend von TAutomatonExpoter verwendet und als physische Datei abgelegt werden.

Das nachfolgende Diagramm zeigt die Hauptklassen des Modelteils von AutoEdit. Aus Platzgründen wurden nur die wichtigsten Methoden und Variablen aufgeführt.



6 Programmierung mit Delphi

6.1 Verarbeitung von XML mit MSXML 4.0

Für die Klasse TAutomatonXML wurde das MSXML 4.0 Framework verwendet welches die Funktionalität für das Parsen von XML Dateien wie auch das Validieren von XML Schemas bereitstellt.

Die Verwendung von MSXML ist praktisch Programmiersprachen unabhängig und würde in C++ oder anderem vergleichbar aussehen. Das Speichern einer Automaten-Definition als XML Datei wurde in AutoEdit etwa wie folgt implementiert:

```
function TAutomatonXML.SaveToFile (filename : String; SaveLayout : Boolean)
  : Boolean;
var e,e2,e3,e4,e5 : IXMLDOMElement; i,z,w : Integer;
    ConditionUsed,TransitionUsed : Boolean;
    a : IXMLDOMAttribute;
begin
  ConditionUsed := false;

  // XML Dokument erstellen
  XMLParser.XML := CreateOleObject('Msxml2.DOMDocument.4.0') as DOMDocument;
  XMLParser.XML.loadXML('<AUTOMATON/>');

  // XML Element erstellen + Attribute
  e := XMLParser.XML.createElement('TYPE');
  a := XMLParser.XML.createAttribute('value');
  a.text := Automaton.AutomatonType;
  e.setAttributeNode(a);
  // Element an das Dokument anhängen
  XMLParser.XML.documentElement.appendChild(e);

  e := XMLParser.XML.createElement('ALPHABET');
  for i := 0 to high(Automaton.Automaton.Alphabet) do begin
    e2 := XMLParser.XML.createElement('ITEM');
```

```

a := XMLParser.XML.createAttribute('value');
a.text := Automaton.Automaton.Alphabet[i].Value;
e2.setAttributeNode(a);
e.appendChild(e2);
end;
XMLParser.XML.documentElement.appendChild(e);
...

```

Der vorangehende Codeausschnitt zeigt das Anlegen eines XML Dokuments sowie das Anlegen der Elemente für den Automantentyp als auch das Alphabet.

Das Laden eines solchen XML Dokuments erfolgt über mehrere Prozeduren, wobei jeweils eine Prozedur für das Einlesen einer XML Ebene verantwortlich ist.

```

function TAutomatonXML.LoadFromFile (filename : String) : Boolean;
var s : String; Schema : XMLSchemaCache;
    Layouted : Boolean; f: file of Byte; XMLF : TStringList;

// Liest die erste Ebene ein - alle Elemente des Rootelements
procedure ReadAutomaton (Element : IXMLDOMNode);
var i : Integer;
begin
for i := 0 to Element.childNodes.length-1 do begin
if Element.childNodes.item[i].nodeName = 'TYPE' then
    Automaton.Automaton.AutomatonType :=
        Element.childNodes.item[i].attributes.getNamedItem('value').text;
if Element.childNodes.item[i].nodeName = 'INITIALSTATE' then
    Automaton.Automaton.StartStatement :=
        Element.childNodes.item[i].attributes.getNamedItem('value').text;
if Element.childNodes.item[i].nodeName = 'TAPEINITIALCHAR' then
    Automaton.Automaton.InitChar :=
        Element.childNodes.item[i].attributes.getNamedItem('value').text;
if Element.childNodes.item[i].nodeName = 'STACKINITIALCHAR' then
    Automaton.Automaton.InitChar :=
        Element.childNodes.item[i].attributes.getNamedItem('value').text;
if Element.childNodes.item[i].nodeName = 'ALPHABET' then
    ReadAlphabet (Element.childNodes.item[i]);
if Element.childNodes.item[i].nodeName = 'STACKALPHABET' then
    ReadStackAlphabet (Element.childNodes.item[i]);
if Element.childNodes.item[i].nodeName = 'TAPEALPHABET' then
    ReadTapeAlphabet (Element.childNodes.item[i]);
if Element.childNodes.item[i].nodeName = 'STATE' then
    ReadStatement (Element.childNodes.item[i]);
if Element.childNodes.item[i].nodeName = 'LAYOUT' then

```

```
        ReadLayout (Element.childNodes.item[i]);
    end;
end;
/////////////////////////////////////////////////////////////////
procedure ReadAlphabet (Element : IXMLDOMNode);
var i : Integer;
begin
    for i := 0 to Element.childNodes.length-1 do begin
        if Element.childNodes.item[i].nodeName = 'ITEM' then
            Automaton.AddAlphabetItem(
                Element.childNodes.item[i].attributes.getNamedItem('value').text);
        end;
    end;
    ///////////////////////////////////////////////////////////////////
    ...
end;
```

In jeder dieser Teilprozeduren wird eine Schleife durchlaufen in der alle Elemente der entsprechenden Ebene durchlaufen werden. Diese Methode ist leicht nachvollziehbar und erlaubt eine einfache Erweiterung der XML Definition bei Bedarf.

6.2 Ausgesuchte Codebeispiele

6.2.1 Überprüfung eines Eingabewortes

Bei der Simulation kann der Benutzer ein beliebiges Eingabewort festlegen welches vom Automaten bearbeitet werden soll. Dies ist jedoch nur sinnvoll, wenn das Wort auch nur aus Zeichen des Eingabealphabets besteht. Bevor die Simulation startet, wird die folgende Überprüfung vorgenommen:

```
function TAutomaton.ValidateInput (s : String) : Boolean;
var i : Integer; a,found : String;
begin
  result := false;
  while (s <> '') do begin
    while (s <> '')and(s[1] = ' ') do delete(s,1,1); // delete spaces
    if s = '' then exit;
    found := '';
    for i := 0 to high(Automaton.Alphabet) do begin
      a := Automaton.Alphabet[i].Value;
      if copy(s,1,length(a)) = a then
        if length(found) < length(a) then found := a;
    end;
    if found <> '' then delete(s,1,length(found)) else exit;
  end;
  result := true;
end;
```

Leerzeichen werden ignoriert womit das Wort "A B C" identisch ist mit "ABC", dies erleichtert die Eingabe von Wörtern, wenn das Eingabealphabet ebenfalls ganze Worte beinhaltet - etwa "BEGIN 1 + 2 END" statt "BEGIN1+2END". Zu beachten ist, daß auch ein Unterschied zwischen "a" und "A" besteht.

6.2.2 Snapping für den visuellen Entwurf

Eine Hilfestellung für das Editieren von Zustandsübergangsgraphen ist das sogenannte Snapping. Beim Verschieben der Symbole im Entwurf mit der Maus ist es ohne diese Hilfe schwer beispielsweise zwei Zustände exakt auf einer gedachten Linie auszurichten. AutoEdit bietet im Vergleich zu JFLAP zwar auch die Möglichkeit die Position pixelgenau als Zahl einzugeben was jedoch weniger komfortabel ist als das Snapping. Wenn mit der Maus Elemente des Graphen verschoben werden, wird automatisch versucht sich an benachbarten Elementen auszurichten um ein gutes visuelles Ergebnis zu erhalten. Realisiert wird diese Funktion mit entsprechenden Prozeduren für die einzelnen Elemente (Zustände, Übergänge und Labels). Es folgt ein Codebeispiel für das Ausrichten der Zustände:

```
function SnapStatement (Range : Integer; p:pStatement) : Boolean;
var i : Integer;
begin
  result := false;
  with Automaton.Automaton do
  for i := 0 to high(Statements) do begin
    if (Statements[i] <> p)and
      (Statements[i].Layout.y+Statements[i].Layout.Radius div 2-Range
        <= p.Layout.Y+p.Layout.Radius div 2)and
      (Statements[i].Layout.y+Statements[i].Layout.Radius div 2+Range
        >= p.Layout.Y+p.Layout.Radius div 2)then begin
        Result := true;
        p.Layout.y := Statements[i].Layout.y+
          Statements[i].Layout.Radius div 2-
          p.Layout.Radius div 2;
        continue;
      end;
    if (Statements[i] <> p)and
      (Statements[i].Layout.x+Statements[i].Layout.Radius div 2-Range
        <= p.Layout.x+p.Layout.Radius div 2)and
      (Statements[i].Layout.x+Statements[i].Layout.Radius div 2+Range
        >= p.Layout.x+p.Layout.Radius div 2)then begin
        Result := true;
        p.Layout.x := Statements[i].Layout.x+
          Statements[i].Layout.Radius div 2-
          p.Layout.Radius div 2;
        continue;
      end;
    end;
  end;
end;
```

Das Snapping erfolgt separat für die X bzw. Y Ausrichtung. Somit ist es möglich, bei 3 Zuständen ein Dreieck auszurichten. Der dritte Zustand richtet sich dann nach der X Koordinate des ersten Zustands und nach der Y Koordinate des Zweiten. Vergleichbar sieht die Funktion für das Snapping von Übergängen aus.

6.2.3 Minimieren eines NEA's

Im Gegensatz zu DEA's gibt es kein berechenbares minimal NEA. Es ist jedoch möglich nach verschiedenen Kriterien dennoch zu minimieren. AutoEdit versucht zunächst alle Zustände zu entfernen die keine Übergänge zu ihnen haben. Zum Startzustand kann durchaus kein Übergang existieren aber der Zustand wird dennoch zwingend benötigt. Deshalb wird der Startzustand von dieser Regel ausgeschlossen.

Ein weiteres Kriterium sind Zustände die selbst keine Endzustände sind und auch keine Übergänge von ihnen weg besitzen. Auch hier bildet der Startzustand eine Ausnahme. Diese beiden Regeln werden solange angewendet bis keine Änderung am Automaten mehr eintritt.

```

procedure TAutomatonTransformation.TransformMinimizeNEA;
var i,z : Integer; State,State2 : pStatement;
    hasTransitionto,NoChange : Boolean;
    DelStates : Array of PStatement;
begin
  Automaton.DeleteSelfEpsilonTransitions;
  Repeat
  // Delete Statements with no Transition to it //
  SetLength(DelStates,0);
  for i := 0 to high(Automaton.Automaton.Statements) do begin
    state := Automaton.Automaton.Statements[i];
    hasTransitionto := false;
    for z := 0 to high(Automaton.Automaton.Statements) do begin
      state2 := Automaton.Automaton.Statements[z];
      hasTransitionto := Automaton.FindTransition(state2,state.name) <> nil;
      if hasTransitionto then break;
    end;
    if not hasTransitionto and
      (State.name <> Automaton.Automaton.StartStatement) // Startstate
    then begin
      SetLength(DelStates,high(DelStates)+2);
      DelStates[high(DelStates)] := state;
    end;
  end;
end;

NoChange := high(DelStates)<0;

for i := 0 to high(DelStates) do
  Automaton.DeleteStatement(DelStates[i].Name);

// Delete none final Statements with no transitions on them //
SetLength(DelStates,0);

```

```
for i := 0 to high(Automaton.Automaton.Statements) do begin
  state := Automaton.Automaton.Statements[i];
  hasTransitionto := high(State.Transitions)>= 0;
  if not hasTransitionto and
    (State.name <> Automaton.Automaton.StartStatement)and // Startstate
    not State.EndStatement then begin
    SetLength(DelStates,high(DelStates)+2);
    DelStates[high(DelStates)] := state;
  end;
end;

NoChange := NoChange or (high(DelStates)<0);

for i := 0 to high(DelStates) do
  Automaton.DeleteStatement(DelStates[i].Name);

until NoChange;
end;
```

Wie man sieht, werden immer erst alle Zustände auf eine Regel überprüft und dabei die betreffenden herausgesucht. Anschließend werden alle Zustände in "DelStates" gelöscht. In Delphi 5 sind dynamische Arraylisten etwas umständlich zu verwalten - Sprachen wie C# bieten inzwischen deutlich einfachere Lösungen an.

6.2.4 Generieren von Scheme Code für DEA

AutoEdit bietet die Möglichkeit Quellcode für die Programmiersprache Scheme zu erzeugen. Umgesetzt wurde dies mit einer fest programmierten Schablone die nur noch mit den entsprechenden Informationen des aktuellen Automaten aufgefüllt wird. Dabei ist die Prozedure für den DEA Code die am wenigsten aufwendige.

```

procedure TAutomatonSchemeCode.GenerateDEACode;
var i,z,t : Integer;
begin

Lines.Clear;
with Lines do begin
Add(';;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;');
Add(';; DFA');
Randomize;
Add(';; example: (DFA ''('+Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+' '+
Automaton.RandomAlphabetItem+''))');
Add(';;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;');
Add('(define DFA');
Add(' (lambda (input)');
Add(' (letrec');

with Automaton.Automaton do
for i := 0 to high(Statements) do begin
if i = 0 then
Add(' (('+Statements[i].Name+' (lambda (tape)') else
Add(' ('+Statements[i].Name+' (lambda (tape)');
Add(' (printf "~a: ~a~n" '''+Statements[i].Name+
' (tape ''(show)))');
Add(' (case (tape ''(read)))');

for z := 0 to high(Statements[i].Transitions) do begin
for t := 0 to high(Statements[i].Transitions[z].Conditions) do begin
Add(' (('+Statements[i].Transitions[z].Conditions[t].Value+
') (tape ''(right!)) ('
+Statements[i].Transitions[z].Statement+' tape)))');
end;
end;
if Statements[i].EndStatement then
Add(' (($ #t)');

```

```

Add('          (else #f))))');
end;

Add('      (DFATape (lambda ()');
Add('          (let ((left-part ''($))');
Add('              (right-part ''($))');
Add('          (lambda (message)');
Add('              (case (car message)');
Add('                  ((init!));
Add('                  (set! left-part (reverse (caadr message)));
Add('                  (set! right-part (cadadr message)));
Add('              ((right!));
Add('              (set! left-part (cons (car right-part) left-part));
Add('              (if (not (equal? right-part ''($)))');
Add('                  (set! right-part (cdr right-part)))));
Add('              ((show));
Add('              (list (reverse left-part) right-part));
Add('              ((read));
Add('              (car right-part));
Add('              (else (error ''tape "~a" (car message))))))));
Add('      (let ((t (DFATape)));
Add('          (t (list ''init! (list (list ''$) (append input (list ''$))))));
Add('          ('+Automaton.Automaton.StartStatement+ t)))));
Add(';;;;;;;;;;;;');
Add('');

```

Das Grundgerüst besteht zum einen aus dem festen DFATape welches das virtuelle Band verkörpert und zum anderen aus den Zuständen und ihren Übergängen die entsprechend generiert werden. Zu beachten bleibt das Scheme nicht in der Lage ist jeden Zustandsnamen zu verarbeiten. Ein Zustand mit dem Namen "Zustand 8" würde auf Grund des Leerzeichens zu Problemen führen. Der Anwender ist deshalb selbst dazu verpflichtet entsprechende Alphabete und Zustandsnamen zu wählen.

6.3 Anbindung an externe Tools

Bereits bei AutoEdit enthalten ist ein Browsertool, welches dem Anwender erlaubt ein beliebiges Verzeichnis nach gültigen Automaten zu durchsuchen und diese in einer übersichtlichen Bildvorschau darzustellen. Hierfür wird AutoEdit im Hintergrund verwendet, um diese Darstellung in Echtzeit aus den XML Dateien zu erzeugen.

Wie bereits in vorangehenden Abschnitten angesprochen, gibt es in AutoEdit externe Simulationsmöglichkeiten. Für jeden beliebigen Automaten kann ein äquivalenter Quellcode der Programmiersprache Scheme erzeugt werden. (Der erzeugte Code wurde mit ChezScheme und Dr. Scheme als Interpreter getestet.) Besonders günstig erweist sich die Verwendung von SchemeEdit welches sich in die Entwicklungsumgebung von AutoEdit integriert und nach erfolgreicher Quellcodeerzeugung automatisch aufgerufen wird, wenn dies beim Anwender installiert ist.

Ein weiteres externes Tool stellt der Visual Compiler Compiler¹ (kurz VCC) dar. AutoEdit bietet die Möglichkeit einen erstellten Automaten in eine Grammatik für VCC zu überführen.

Beide Exportfunktionen dienen der Simulation des Akzeptanzverhaltens eines Automaten. Der Scheme-Automat kann dabei mit beliebigen Eingabewörtern auf Akzeptanz getestet werden und liefert dabei auch die Abarbeitungsschritte in Textform zurück. Ein VCC Compiler bietet darüber hinaus einen Scanner der sowohl von Eingabedateien als auch Strings zunächst einen Token-Value-Strom bildet der anschließend vom generierten deterministischen Kellerautomaten verarbeitet werden kann.

¹Ein Werkzeug zur Erstellung von Compilern an Hand einer Grammatik. Verwand mit LEX und YACC aus der Unixwelt jedoch mit Scheme bzw. C# als Zielformat.

7 Weiterentwicklungsmöglichkeiten

Im Vergleich zu JFLAP konzentriert sich AutoEdit auf Automaten und deren Darstellung. Grammatiken und deren Transformation werden nicht angeboten. Dieser Komplex wäre eine sinnvolle Erweiterung. Durch FADL würden diese Funktionen auch nicht in AutoEdit selbst integriert werden müssen, sondern als externes Tool oder als Bestandteil von VCC eingebracht werden können.

Derzeit ist es nicht möglich, vorhandene Automaten miteinander zu vereinen um beispielsweise wiederkehrende Teilautomaten modular zu speichern. Diese Funktionalität könnte in den AutoEditBrowser eingebracht werden, da man hier übersichtlich aus vorhandenen Automaten wählen kann. Funktionen zum Vergleichen von Automaten wären an dieser Stelle ebenfalls denkbar.

AutoEdit ist nicht in der Lage Layouteinstellungen von vorhandenen Automaten oder Zuständen zu übernehmen. Will man beispielsweise alle Zustände blau einfärben, so muss man dies separat für jeden einzelnen Zustand editieren, was sehr zeitaufwendig ist. Hier wäre eine Art Stylesheet für das Layout sinnvoll, um diesen Aufwand zu reduzieren.

8 Fazit

Die Aufgaben, ein Werkzeug für die Publikation von Automaten zu entwickeln, wurde mit AutoEdit gelöst. Es bestehen natürlich immer noch Erweiterungsmöglichkeiten aber die ursprünglich geforderte Funktionalität ist vorhanden. AutoEdit wurde auch für Lernzwecke, mit der Zielstellung Schülern die Automatentheorie auf praktische Art und Weise zu verdeutlichen, konzipiert. Aus ersten Erfahrungsberichten von Anwendern geht hervor, dass AutoEdit auch diesem Ziel gerecht werden kann.

Bei der Entwicklung eines Tools wie AutoEdit ist man gezwungen sich eingehender mit der Automatentheorie zu beschäftigen als das man dies im normalen Studium tun würde. Durch die Entwicklung von VCC und den damit verbundenen Überlegungen zum Compilerbau, ergibt sich ein fundiertes Wissen zu diesen Themengebieten. Da man bei der Softwareentwicklung häufig die Funktionalität von Scanner und Parser benötigt, ist dieses Wissen uneingeschränkt jedem Entwickler zu empfehlen.

9 Anhang

9.1 XML Schema für Automaten

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="AUTOMATON">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TYPE">
          <xs:complexType>
            <xs:attribute name="value" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="DEA"/>
                  <xs:enumeration value="NEA"/>
                  <xs:enumeration value="DKA"/>
                  <xs:enumeration value="NKA"/>
                  <xs:enumeration value="TM"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
        <xs:element name="ALPHABET">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ITEM"
                minOccurs="0"
                maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="value"
                    type="xs:string"
                    use="required"/>
                </xs:complexType>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

```
        </xs:element>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="STACKALPHABET"
    minOccurs="0">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ITEM"
                minOccurs="0"
                maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="value"
                        type="xs:string"
                        use="required"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="TAPEALPHABET"
    minOccurs="0">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ITEM"
                minOccurs="0"
                maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="value"
                        type="xs:string"
                        use="required"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="STATE"
    minOccurs="0"
    maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="TRANSITION"
```

```
        minOccurs="0"
        maxOccurs="unbounded">
<xs:complexType>
  <xs:sequence>
    <xs:element name="LABEL"
      minOccurs="0"
      maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="read"
          type="xs:string"
          use="required"/>
        <xs:attribute name="direction"
          use="optional">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="RIGHT"/>
              <xs:enumeration value="LEFT"/>
              <xs:enumeration value="NONE"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="write"
          type="xs:string"
          use="optional"/>
        <xs:attribute name="topofstack"
          type="xs:string"
          use="optional"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="target"
    type="xs:string"
    use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="name"
  type="xs:string"
  use="required"/>
<xs:attribute name="finalstate"
  type="xs:string"
  use="optional"/>
```



```
        use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="RADIUS" minOccurs="0">
    <xs:complexType>
        <xs:attribute name="value"
            type="xs:unsignedInt"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="COLOR" minOccurs="0">
    <xs:complexType>
        <xs:attribute name="value"
            type="xs:string"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="PENWIDTH" minOccurs="0">
    <xs:complexType>
        <xs:attribute name="value"
            type="xs:unsignedInt"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="PENCOLOR" minOccurs="0">
    <xs:complexType>
        <xs:attribute name="value"
            type="xs:string"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="FONTSIZE" minOccurs="0">
    <xs:complexType>
        <xs:attribute name="value"
            type="xs:unsignedInt"
            use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="FONTCOLOR" minOccurs="0">
    <xs:complexType>
        <xs:attribute name="value"
            type="xs:string"
```

```
        use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="TRANSITIONLAYOUT"
    minOccurs="0"
    maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="LEFT"
                minOccurs="0">
                <xs:complexType>
                    <xs:attribute name="value"
                        type="xs:integer"
                        use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="TOP"
                minOccurs="0">
                <xs:complexType>
                    <xs:attribute name="value"
                        type="xs:integer"
                        use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="FONTCOLOR"
                minOccurs="0">
                <xs:complexType>
                    <xs:attribute name="value"
                        type="xs:string"
                        use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="FONTSIZE"
                minOccurs="0">
                <xs:complexType>
                    <xs:attribute name="value"
                        type="xs:unsignedInt"
                        use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="PENWIDTH"
                minOccurs="0">
```

```
<xs:complexType>
  <xs:attribute name="value"
                type="xs:unsignedInt"
                use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="PENCOLOR"
            minOccurs="0">
  <xs:complexType>
    <xs:attribute name="value"
                  type="xs:string"
                  use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="ARROWSIZE" minOccurs="0">
  <xs:complexType>
    <xs:attribute name="value" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="none"/>
          <xs:enumeration value="very small"/>
          <xs:enumeration value="small"/>
          <xs:enumeration value="normal"/>
          <xs:enumeration value="large"/>
          <xs:enumeration value="very large"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="LABELLAYOUT"
            minOccurs="0"
            maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="LEFT"
                  minOccurs="0">
        <xs:complexType>
          <xs:attribute name="value"
                        type="xs:integer"
                        use="required"/>
        </xs:complexType>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

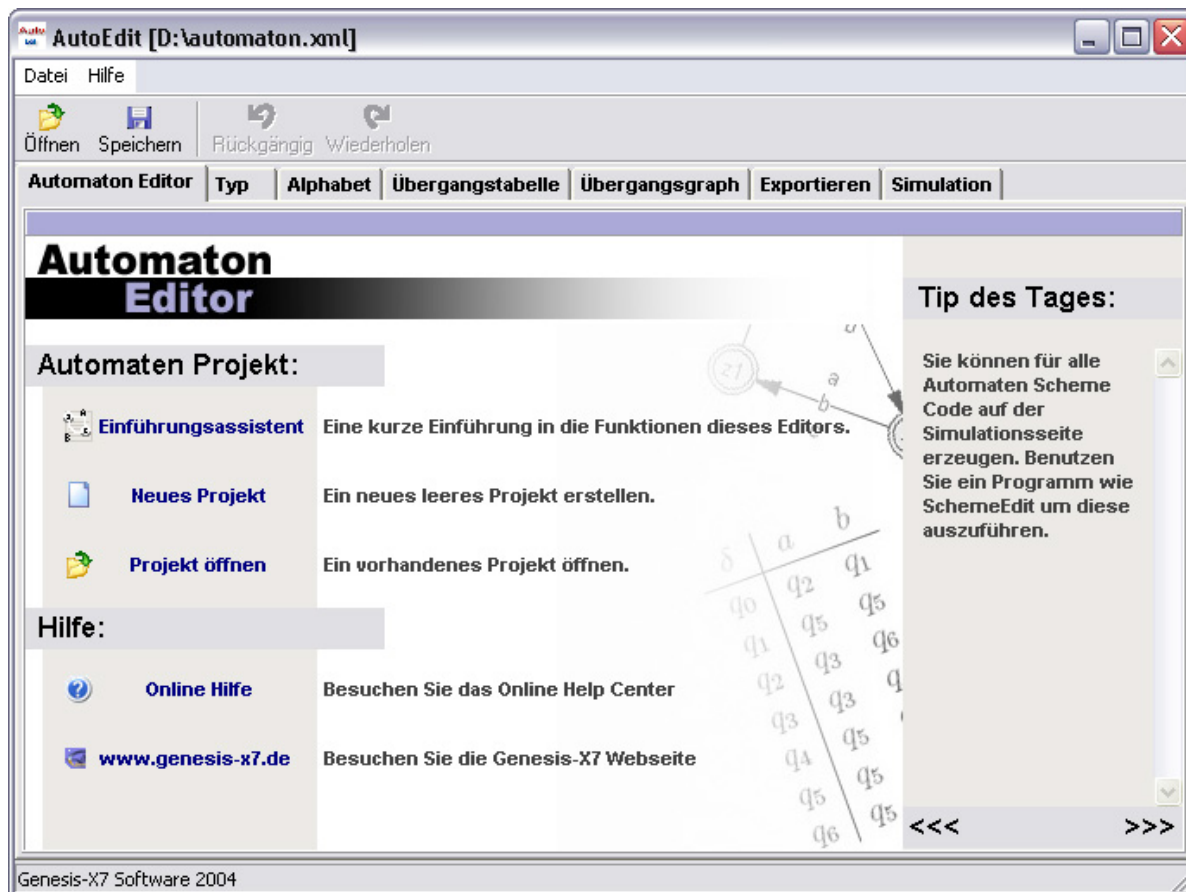
```
</xs:element>
<xs:element name="TOP"
             minOccurs="0">
  <xs:complexType>
    <xs:attribute name="value"
                  type="xs:integer"
                  use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="FONTCOLOR"
             minOccurs="0">
  <xs:complexType>
    <xs:attribute name="value"
                  type="xs:string"
                  use="required"/>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="read"
              type="xs:string"
              use="required"/>
<xs:attribute name="write"
              type="xs:string"
              use="optional"/>
<xs:attribute name="topofstack"
              type="xs:string"
              use="optional"/>
<xs:attribute name="direction"
              type="xs:string"
              use="optional"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="target"
              type="xs:string"
              use="required"/>
<xs:attribute name="drawvertical"
              type="xs:string"
              use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
```

```
        <xs:attribute name="name"
                    type="xs:string"
                    use="required"/>
    </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

9.2 Tutorial

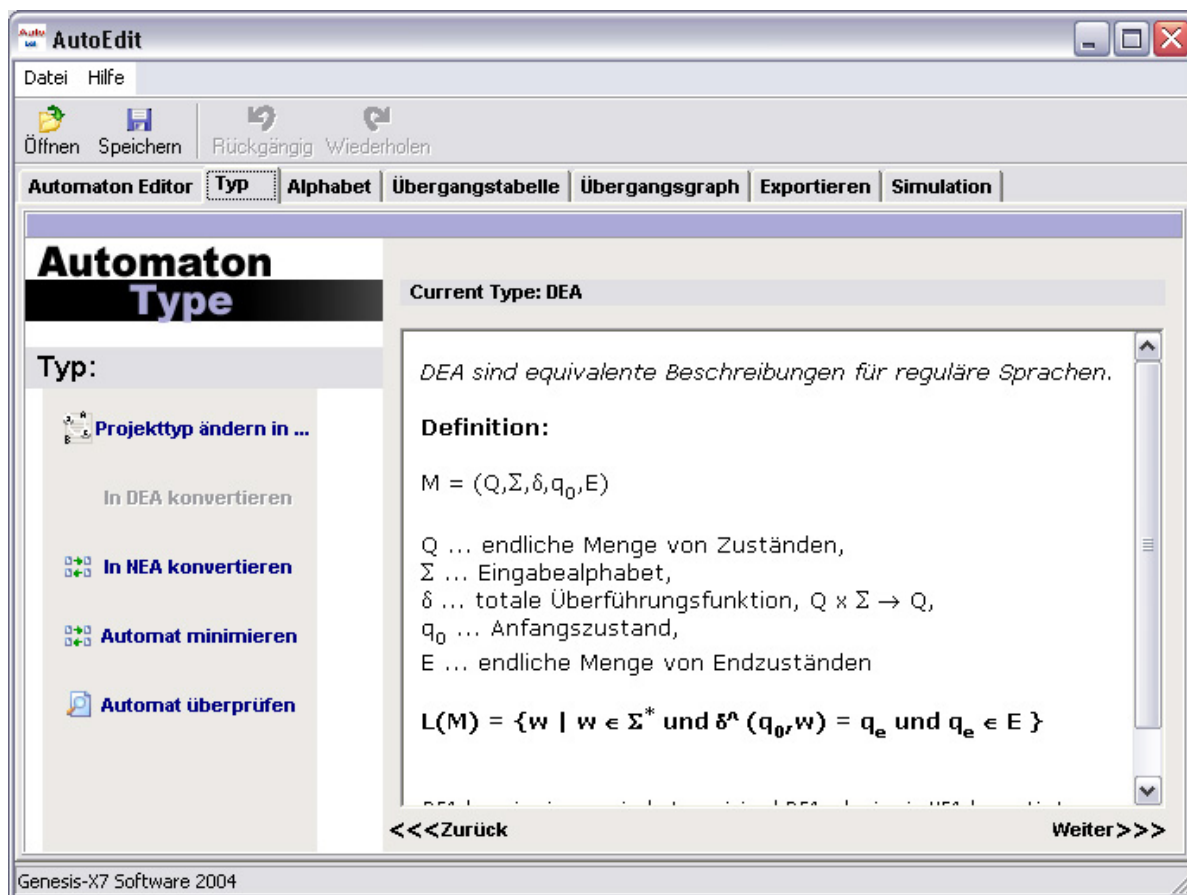
Dieses Tutorial zeigt die Erstellung eines einfachen Automaten mit AutoEdit und die Einbindung in ein LaTeX Dokument. Viele Funktionen wie die Simulation werden hier jedoch nicht beschrieben. Es wird vorausgesetzt das AutoEdit installiert wurde und MSXML 4.0 über die automatische Installation auf dem Computer vorhanden ist.

SCHRITT 1: ANLEGEN EINES NEUES PROJEKTS



Nach dem Start von AutoEdit erscheint das hier dargestellte Fenster. Im Programm selbst wird ebenfalls ein Tutorial angeboten welches über den ersten Button aufgerufen werden kann. Für dieses Tutorial wählen wir "Neues Projekt".

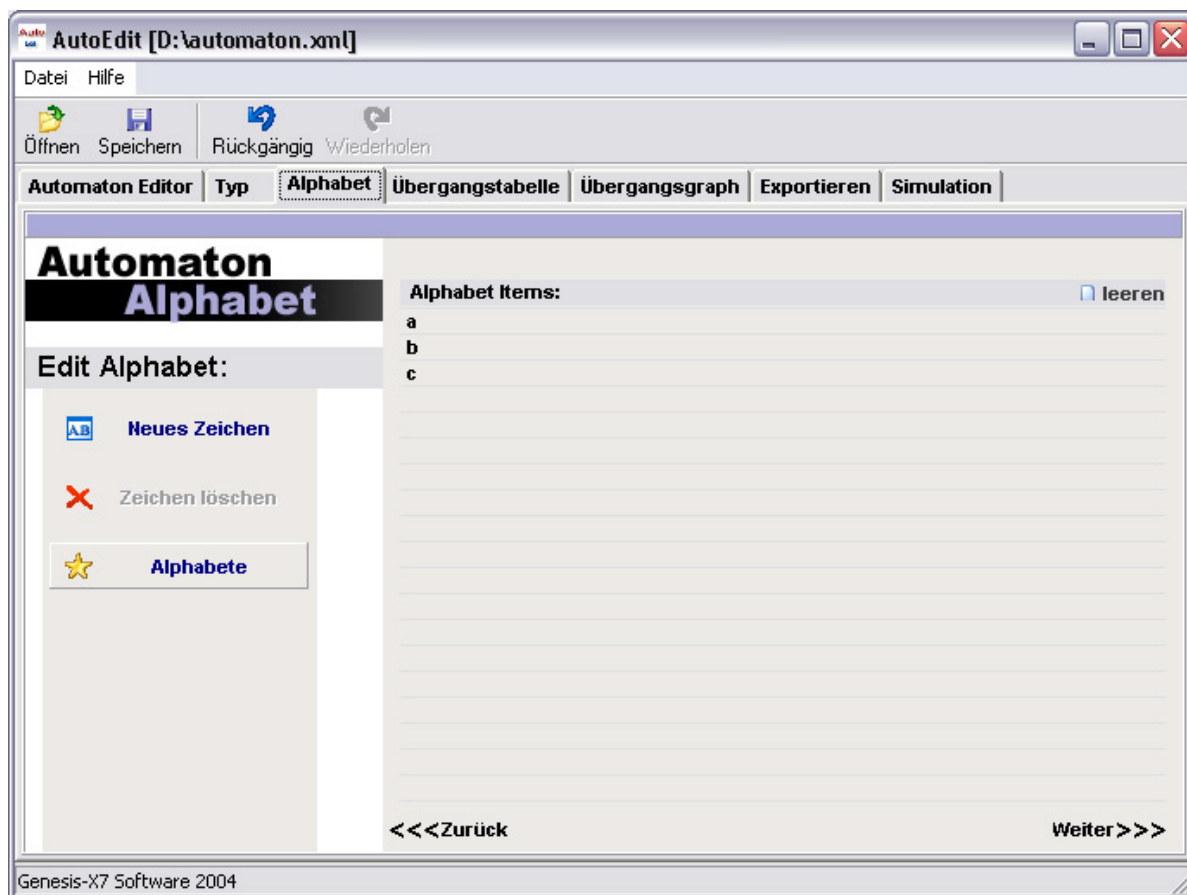
SCHRITT 2: FESTLEGEN DES AUTOMATENTYPES



Ein neues Projekt wird als DEA initialisiert. Um einen anderen Typ zu wählen klickt man auf "Projekttyp ändern in..." und wählt den entsprechenden Typ aus dem erscheinenden Popupmenü aus. Für dieses Beispiel wählen wir NEA aus und klicken anschließend auf "Weiter" unten rechts.

An dieser Stelle kann auch ein vorhandener Automat in einen anderen Typ konvertiert oder minimiert werden.

SCHRITT 3: EINGABEALPHABET FESTLEGEN



Hier wird das Eingabealphabet des Automaten festgelegt. Dies ist zwingend nötig, da ohne ein Alphabet im Nachfolgendem keine Übergänge erstellt werden können. Es ist natürlich möglich jeder Zeit auf diese Seite zurückzukehren und das Alphabet zu erweitern.

Um ein Zeichen zum Alphabet hinzuzufügen, klicken wir auf "Neues Zeichen" und erhalten ein Eingabefeld in der rechten Liste. Für dieses Beispiel erstellen wir die Zeichen "a", "b" und "c". Typische Alphabete können auch über den Button "Alphabete" erstellt werden.

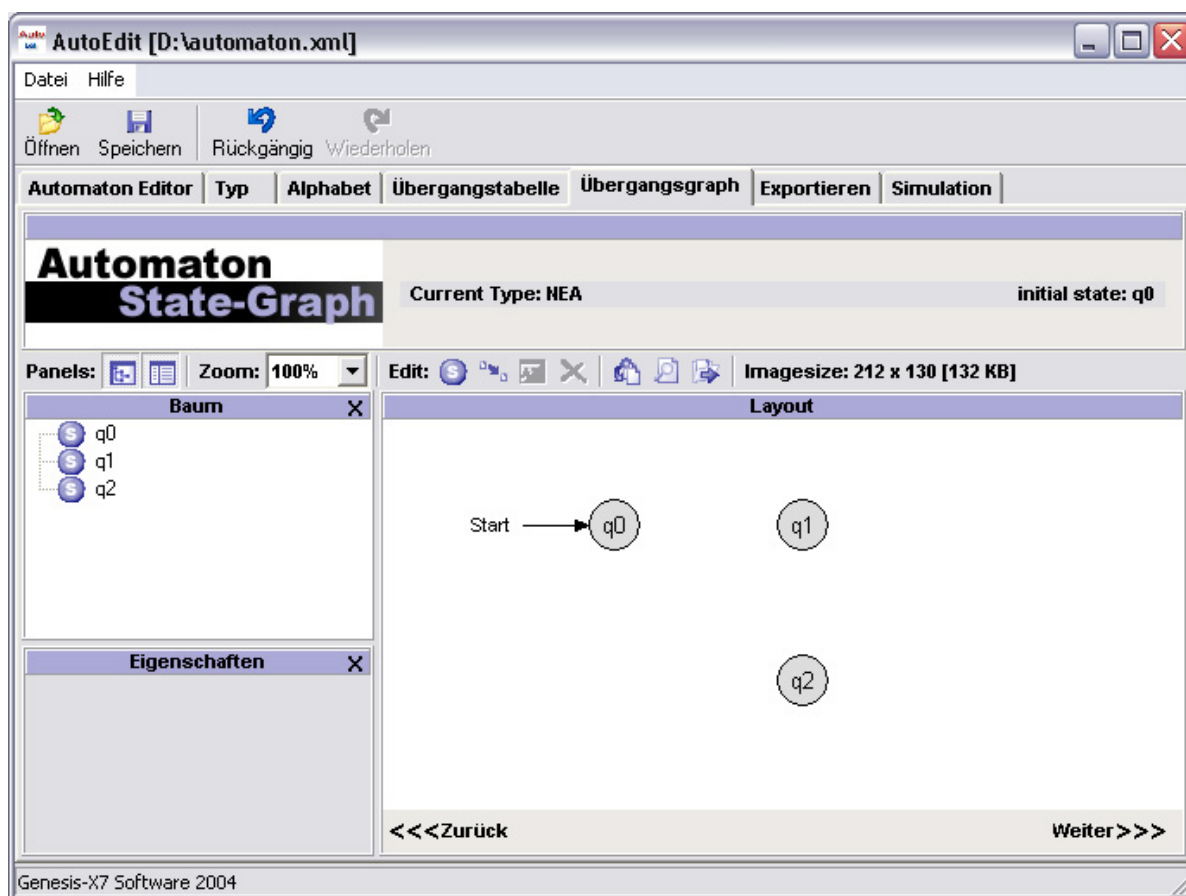
Anschließend wieder auf "Weiter" klicken, um auf die nächste Seite zu gelangen.

SCHRITT 4: GRAPHISCHER ENTWURF - ZUSTÄNDE

Zunächst landen wir auf der Seite "Übergangstabelle". Hier kann der Automat definiert werden ohne den graphischen Aspekt zu betrachten. Da dies aber in diesem Tutorial keine Rolle spielen soll, klicken wir erneut auf "Weiter", um in die graphische Entwurfsansicht zu gelangen.

Zunächst erstellen wir drei Zustände, indem wir auf der Toolbar (Symbolleiste im oberen Bereich) das erste Symbol (kleine Kugel mit einem S) anwählen. Nun an 3 beliebigen Stellen auf der weißen Fläche klicken um jeweils einen Zustand anzulegen. Anschließend erneut das Symbol in der Toolbar anwählen, um diesen Modus wieder zu verlassen.

Das Ergebnis sollte etwa wie folgt aussehen:

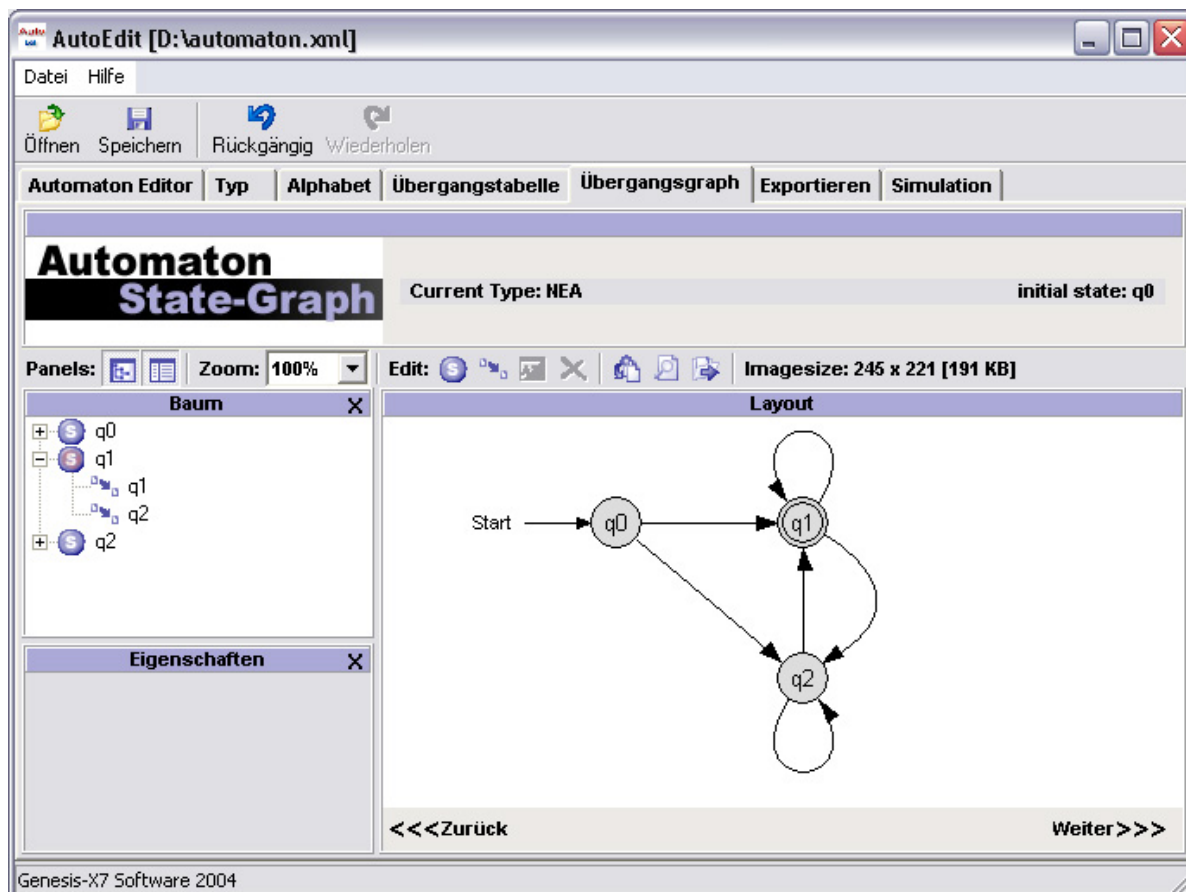


SCHRITT 5: GRAPHISCHER ENTWURF - ÜBERGÄNGE

Für die Übergänge zwischen den Zuständen wählen wir das zweite Symbol aus der Toolbar an. Nun klicken wir den ersten Zustand einmal an und anschließend einmal auf Zustand zwei, um einen Übergang zwischen diesen beiden zu erstellen (1. Klick = Quelle, 2.Klick = Ziel).

Wenn ein Übergang von einem Zustand zum selben Zustand erstellt werden soll, muss dementsprechend zweimal der selbe Zustand angeklickt werden.

Verbinden wir nun die Zustände so, dass folgender Automat entsteht:

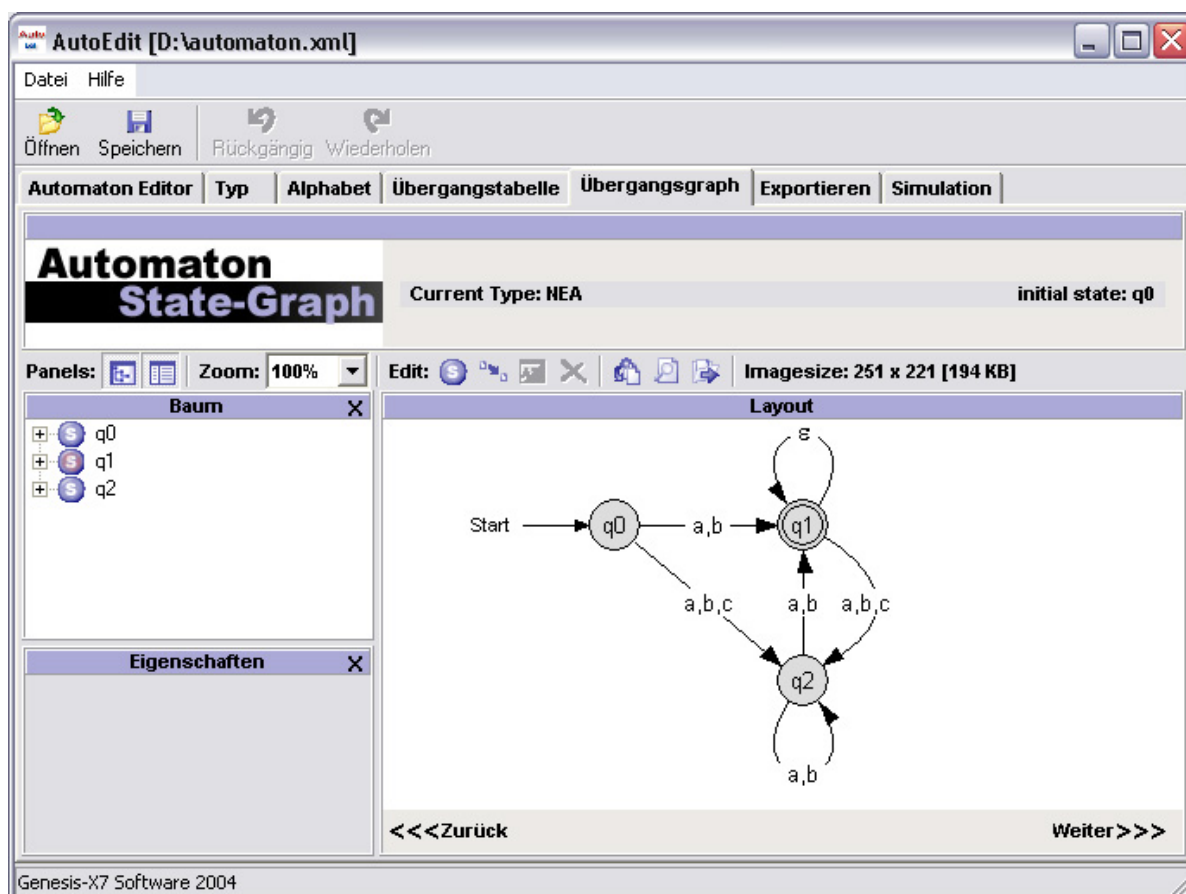


SCHRITT 6: GRAPHISCHER ENTWURF - LABELS

Nun fehlen noch die Labels an den Übergängen - die Alphabetszeichen unter denen der Übergang möglich ist.

Dazu zunächst einen Übergang auswählen (dieser wird nun rot dargestellt) und anschließend auf das dritte Symbol aus der Toolbar (kleine Buchstaben) klicken. Es wird automatisch der erste Buchstabe des Alphabets verwendet. Wenn man erneut auf das Symbol in der Toolbar klickt wird ein weiteres Label erstellt. Dabei versucht AutoEdit den am ehesten zu erwartenden Buchstaben des Alphabets zu wählen. Um aber selbst das Alphabetszeichen festzulegen bedient man sich der Eigenschaftsliste links unten. Das Dropdown "Read" erlaubt die Wahl eines Alphabetszeichens.

Die folgende Darstellung zeigt das Ziel dieses Schritts:

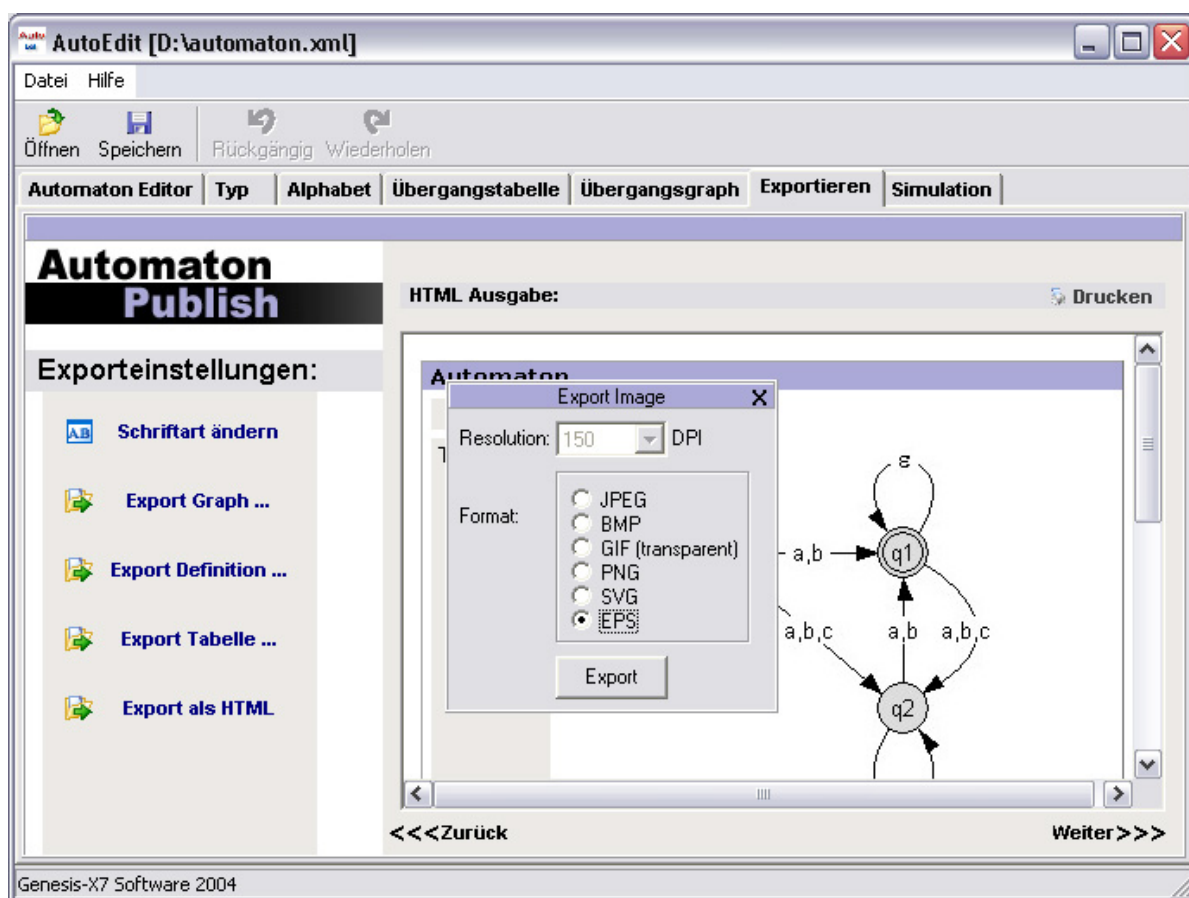


SCHRITT 7: EXPORTIEREN

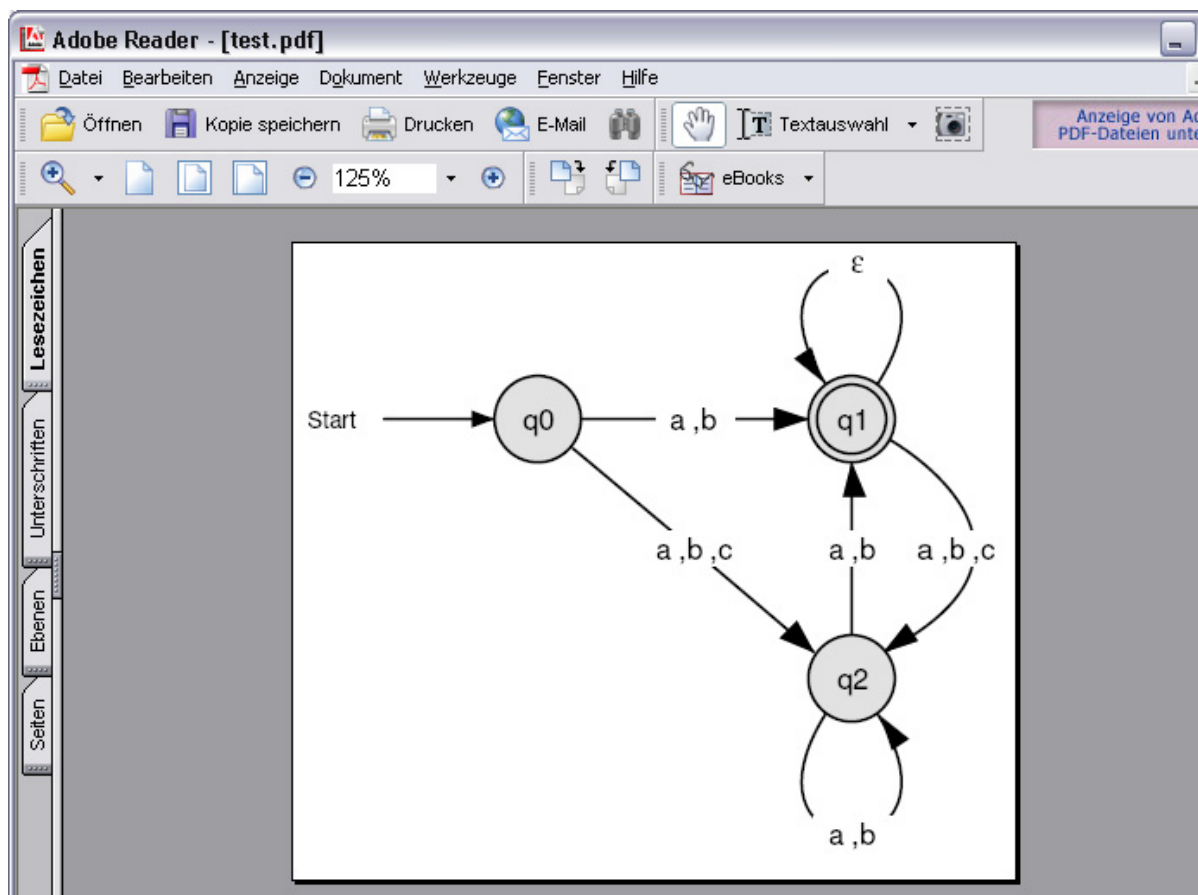
Nachdem der Entwurf des Automaten abgeschlossen ist, klicken wir erneut auf "Weiter", um auf die Export Seite zu gelangen.

Hier können verschiedene Graphiken exportiert werden. "Export Graph" erlaubt das Exportieren des Zustandsübergangsgraphen in verschiedenen Formaten und Auflösungen. Für ein LaTeX Dokument eignet sich besonders EPS bzw. das daraus zu gewinnende Format PDF.

Für dieses Tutorial exportieren wir den Graphen als EPS und beantworten die Frage nach der automatischen Erstellung der zugehörigen PDF Datei mit Ja (Dies wird nur angeboten, wenn auf dem System epstopdf gefunden wurde. Im kostenlosen Packet GNU GhostScript für Windows ist dieses Tool mit enthalten.) Speichern wir nun den Graph als Test.eps ab:



SCHRITT 8: EINBINDEN IN LATEX



Die PDF Datei kann nun einfach in eine LaTeX Dokument eingebunden werden:

```
\begin{figure}
  \centering
  \includegraphics{Test}
  \label{fig:Test}
\end{figure}
```

Da das EPS / PDF Format ein Vektorformat ist, ist die Druckqualität bei allen Auflösungen sehr gut, womit ein deutlicher Vorteil gegenüber einer JPEG Datei erzielt wird.

9.3 Datenträgerbeschreibung

Auf dem beiliegenden Datenträger befinden sich die Quelldateien für AutoEdit sowie andere Dateien die hier kurz aufgelistet und erklärt werden sollen.

AUTOEDIT DELPHI SOURCE

In diesem Verzeichnis befinden sich die Quelldateien für AutoEdit. Genutzt wurde Borland Delphi 5 - andere Versionen sind leider oftmals nicht 100% kompatibel. Das Einrichten einer passenden Delphi 5 Umgebung ist ebenfalls sehr aufwendig da mehrere Komponenten (in diesem Verzeichnis enthalten) installiert werden müssen.

INNO SETUP

Dieses Freewaretool erstellt auf einfache Art und Weise Setups für Applikationen. Im Quellcodeverzeichnis von AutoEdit befindet sich eine passende Quelldatei für dieses Tool. Dabei werden sowohl die Beispiele als auch die benötigten Dateien von AutoEdit zu einem Setup verpackt.

GHOST SCRIPT

Für die automatische Umwandlung von EPS ind PDF Dateien wird Ghost Script benötigt. Die Installation ist aber für die Verwendung von AutoEdit nicht erforderlich wenn dieses Feature nicht benötigt wird.

AUTOEDITSETUP

Dies ist die fertige Setupdatei für AutoEdit 1.04 BETA. Diese Installation sollte auf jedem Windows Betriebssystem möglich sein (getestet wurde AutoEdit vorwiegend unter Windows XP).

DOCUMENTATION

LaTeX Dateien und PDF von diesem Dokument.

10 Literaturverzeichnis

- [JFLAP] *Java Formal Languages and Automata Package*
<http://www.cs.duke.edu/~rodger/tools/tools.html>
Susan H. Rodger, 2004
- [FLAT] *FLAT-Werkzeuge*
<http://www.inf.hs-zigr.de/~wagenkn/TI/Automaten/FLAT-Software/>
Stand: 11.11.2003
- [Vaucanson-G] *Vaucanson-G: A package for drawing automata and graphs*
<http://www.liafa.jussieu.fr/~lombardy/Vaucanson-G/>
Jacques de Vaucanson, 2003
- [Vorlesungsskript] *Vorlesungsskript „Theorie der formalen Sprachen und Automatentheorie“*
Prof. Dr. Christian Wagenknecht, 2001